HASH FUNCTIONS IN CRYPTOGRAPHY

Master of science thesis

Joseph Sterling Grah

Institutt for Informatikk
Det matematisk-naturvitenskapelige fakultet

Universitet i Bergen

June 1, 2008

AKNOWLEDGMENTS

It is a pleasure to thank the many people who made this thesis possible.

I state my gratitude to my supervisor, Professor Igor Semaev. With his enthusiasm, and his great efforts to explain things clearly and simply. Throughout my thesis-writing period, he provided sound advice and lots of good ideas.

I wish to thank all my friends and my extended family (brothers, sisters, brothers-in-law, sisters-in-law, cousins, aunts, and uncles) for providing a loving environment for me.

I am grateful to Jarle, for his support and encouragements. A special thank goes also to all my former and actual colleagues for their kind support.

I wish to thank my parents, Henriette and Antoine. They bore me, raised me, supported me, taught me, and loved me.

Lastly, and most importantly, I want to thank my daughter "prinsesse Kelly", her happy mood always inspires me, and my girlfriend "Lo" for loving me, encouraging me, and supporting me unconditionally.

To them, I dedicate this thesis.

ABSTRACT

This thesis is concerned with giving both an overview of the application of hash functions in cryptography and a presentation of today's standard cryptographic hash functions.

Cryptographic hash functions are a valuable tool in cryptography. They are applied in many areas of information security to provide protection of the authenticity of messages; data integrity verification which prevents modification of data from going undetected, time stamping and digital signature scheme.

Contents

Part I
Introduction

1 Introduction

Cryptographic hash functions cannot be thought of outside mathematics. In fact, computer science is a science due to mathematics; in other word, it is because of its mathematical properties that computer science can be explained and understood like any other scientific knowledge. This is why we have decided to introduce the reader not only to some fundamental concepts in mathematics but also to some terminologies that will be used later in this paper. Knowing that the words which make up our languages can be prone to different interpretations, we have introduced this section as a reference that should define and clarify some few technical words which might be susceptible to lead to any ambiguity whatsoever.

1.1 Goal

The goal of this paper is to introduce the reader to hash functions and their area of application. Some standards hash functions are presented in detail to give a more in-depth explanation of how most cryptographic hash functions are designed.

1.2 How is the paper organized?

This paper is organized in 4 parts as follows:

Part 1, introduces the content of this paper and explains how it is organized.

Part 2, defines some concepts we have judged important to understand in order to fully take advantage of what is presented later in the text. Hash functions in cryptography are defined and a discussion of the idea underlying the birthday paradox is elaborated.

Part 3, gives an overview of the main areas where cryptographic hash functions are applied.

Part 4, presents the main building blocks of the standard hash functions and also introduces three well known hash functions which are used worldwide.

Part II
Definition of concepts

2 Definition of concepts

2.1 Sets

Sets

Definition. A *set* is an unordered collection of distinct objects which share a common property. The objects of a set are called *elements*.

It is important to note that for any object, one should be able to clearly determine whether or not it is an element in the set under consideration. For example let A be the *set* of all positive even integers greater than 2 and strictly less than 8. We can clearly determine that the integer 4 is an *element* of the *set* A, but not the integer 8.

Subsets

Definition. A *subset* is a *set* which is contained in another *set*. That is, A is a *subset* of B if every *element* of the *set* A is also an *element* of the *set* B, and we write $A \subseteq B$. In case B contains an *element* that is not in A, we say that A is a *proper subset* of B, and we write $A \subset B$.

Finite sets

Definition. A *set* A is *finite* if it contains a finite number of *elements*.

An *infinite set* contains an infinite number of *elements*.

For any finite *set* A, $|A|$ denotes the number of *elements* in A, and it is known as the *cardinality* of A.

We will not go any deeper in this interesting subject of mathematics called **set theory**. We direct the reader to any writing on set theory to learn more about this fundamental concept of mathematic which provides the support needed by other mathematical topics in order to be concisely formulated and understood.

2.2 Cartesian Products

Cartesian Products

Definition: Let A and B be two sets, the *cartesian product* or *cross product*, of A and B denoted by $A \times B$ equals $\{(a, b) \mid a \in A \text{ and } b \in B\}$.

The elements of $A \times B$ are ordered pairs. The definition of the cartesian product can be extended to more than two sets. For instance, if we consider the sets A, B and C, we say that $A \times B \times C$ equals $\{(a, b, c) \mid a \in A,\ b \in B \text{ and } c \in C\}$. The elements of $A \times B \times C$ are called ordered triples. When more than 3 sets are involved, say n sets, the elements are called ordered n-tuples.

Example: Let $A = \{2, 3\}$, $B = \{4, 5\}$, $C = \{6\}$, the following statements hold true:

- $A \times B = \{(2, 4), (2, 5), (3, 4), (3, 5)\}$
- $B \times A = \{(4, 2), (4, 3), (5, 2), (5, 3)\}$
- $B^2 = B \times B = \{(4, 4), (4, 5), (5, 4), (5, 5)\}$
- $A \times B \times C = \{(a, b, c) \mid a \in A,\ b \in B \text{ and } c \in C\}$, for instance $(2, 4, 6)$ is an ordered triple of $A \times B \times C$.

2.3 Relations

Relations

Definition: For two sets A and B, any subset of $A \times B$ is called a *relation* from A to B. Any subset of $A \times A$ is called a binary *relation* on A.

Example: Consider the sets A and B from the example above, the followings are relations from A to B:

- $\{(2, 5)\}$
- $\{(2, 4), (3, 5)\}$
- $A \times B$

Thus, a relation can simply be defined as a set of ordered pairs; where the first elements in the ordered pairs form the *domain*, whereas the second elements in the ordered pairs form the *range*.

2.4 Function

Function

Definition. Consider two nonempty sets A and B. A function f from A to B, denoted $f : A \rightarrow B$ is a relation from A to B where each element of the *set* A is assigned to a unique *element* of the *set* B.

We write $f(a) = b$ whenever (a, b) is an ordered pair in the relation defined by the function f, and b is called the *image* of a under f, whereas a is called the *preimage* of b. Notice that not all relations from the *set* A to B are functions.

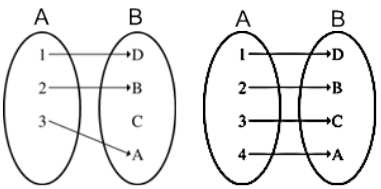
Example: Let $A = \{1, 2, 3\}$, $B = \{k, l, m\}$,

- $\star$ $f = \{(1, k), (2, m), (3, m)\}$ is a function which implies that it is a relation from A to B
- $\star$ $r_1 = \{(1, k), (2, m)\}$ is a relation but not a function as the element 3 in the *set* A is not assigned to any *element* in the *set* B.
- $\star$ $r_2 = \{(1, k), (2, l), (2, m), (3, m)\}$ is a relation but not a function because the element 2 in the *set* A has been assigned to more than one element in the *set* B

One-To-One Function

Definition: a function $f : A \rightarrow B$ is called *one-to-one* or *injective*, if it maps distinct elements from A to distinct elements in B.

A pictorial representation of two injective functions appears in figure 1.



(a) Injection (b) Injection

Figure 1: *Injective functions.*

In other words, if $f : A \rightarrow B$ is injective, then for $a_1, a_2 \in A$, $f(a_1) = f(a_2)$ implies that $a_1 = a_2$. This implication simply means that every *element* of the *codomain* B is mapped onto by one and only one *element* of the *domain* A.

Example: Consider the function $f : \mathbb{R} \rightarrow \mathbb{R}$ where $f(x) = x + 1$ for all $x \in \mathbb{R}$. Thus, for $r_1, r_2 \in \mathbb{R}$, we have

$f(r_1) = f(r_2) \Rightarrow r_1 + 1 = r_2 + 1 \Rightarrow r_1 = r_2$, the function f is *injective* or *one-to-one*.

However, as for the function $g : \mathbb{R} \rightarrow \mathbb{R}$, defined by $g(x) = x^2 - x$ for each real number x, we can easily find that,

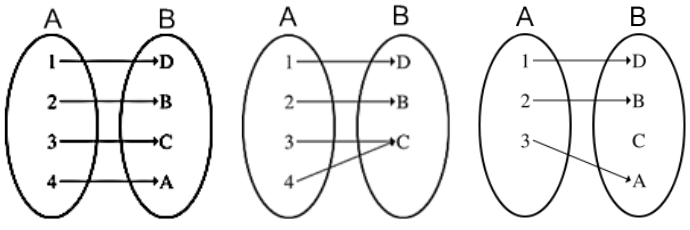
$g(0) = (0)^2 - 0 = 0$ and $g(1) = (1)^2 - (1) = 1 - 1 = 0$, Hence the function g is not injective, since $g(0) = g(1)$ but $0 \neq 1$. This strictly means that g is not injective since we are able to find one *element* in the codomain which is mapped onto by more than one *element* in the domain.

Note that hash functions should guarantee that the latter situation referred to as a **collision** rarely occurs in practice; The credibility of any hash function depends entirely on its ability not to exhibit such behaviour.

Onto Function

Definition: A function $f : A \rightarrow B$ is called *onto*, or *surjective*, if $f(A) = B$.

That is, if for every *element* b in the *codomain* B, there is at least one *element* a in the *domain* A such that $f(a) = b$.



(a) Surjective (b) Surjective (c) Non surjective

Figure 2: *Surjective and non-surjective functions.*

A pictorial representation of two *surjective* functions and one $non - surjective$ function is shown respectively in figure 2(a), 2(b) and 2(c). The function in figure 2(c) is not surjective because there is no element $a \in A$ such that $f(a) = C$.

There exist a wide range of interesting arithmetic functions in mathematics which are applied in computer science as well.
Among them we can list the following:

The additive function: $a + b, a - b$ which equals $a + (-b)$

The multiplicative function: $a \times b, a \div b$ which equals $a \times (1 \div b)$

The power function: $x^n = \underbrace{x \times x \times \cdots \times x}_{n \ times}$, where n is an integer number

The absolute value function: $|x|$, which equals $-x$ if x is negative and x otherwise, i.e. $|5| = 5, |-2| = 2$.

The floor function: $\lfloor x \rfloor$, which is the largest integer less than or equal to x, i.e. $\lfloor 3.6 \rfloor = 3, \lfloor -3.6 \rfloor = -4$.

The ceiling function: $\lceil x \rceil$, is the smallest integer greater than or equal to x, i.e. $\lceil 7.3 \rceil = 8, \lceil -3.4 \rceil = -3$

The trunk function: $trunc(x)$, deletes the fractional part of the real number x, i.e. $trunc(2.95) = 2, trunc(5) = 5$. It appears that this last function is mostly used in pocket calculators and programming languages. One uses the floor function in mathematics as it produces the same result as the function trunc.

The modulus function: $a \ modulo \ b$, is the remainder of the division of a by b, i.e. $7 \ modulo \ 5 = 2$.

2.5 Domain, Co-domain and Range

Domain

Definition: For the function $f : A \rightarrow B$, A is called the domain of f.

Co-domain

Definition: For the function $f : A \to B$, B is called the codomain of f.

Range

Definition: For the function $f : A \to B$, the subset of B consisting of those elements that appear as the second components in the ordered pairs of f is called the *range* of f and is denoted by $f(A)$ since it is the set of images under f.

2.6 Division, Prime Numbers, Integers Modulo n

The Division Algorithm

Definition: Given $a, b \in \mathbb{Z}$, with $b \neq 0$
There exist unique integers q and r such that $a = q \times b + r$ and $0 \leq r \leq |b|$, where $|b|$ denotes the absolute value of b.

q is the *quotient*
r is the *remainder*
b is the *divisor*
a is the *dividend*

We say that b divides a, and we write $b|a$, if there is an integer n such that $a = b \times n$. In this case, b is a divisor of a, and a is a multiple of b.

Example:

$14 = 2 \times 7$, so both 2 and 7 are divisors of 14 which is a multiple of 2 and is also a multiple 7. We also observe that 1 and 14 are divisors of 14, since $14 = 1 \times 14$. In general, every number divides itself, and 1 divises all numbers.

Prime numbers

Definition: A *prime* number is a natural number which is only divisible by two natural numbers, 1 and itself. All other numbers are called *composite* numbers.

Example:

$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$ are the positive *prime* numbers less than 40.

6 is not a *prime* number since 6 is divisible by $1, 2, 3$ and 6. Therefore 6 is called a *composite* number. It is equal to the product of two prime numbers, namely 2×3.

The Integers modulo n

The modulus operation finds the remainder of division of one number by another number.

Definition: Let $n \in \mathbb{Z}^+$, with $n > 1$. Given two numbers $a, b \in \mathbb{Z}$, we say that a is *congruent* to b *modulo* n, and we write $a \equiv b$ *modulo* n, if n divides $(a - b)$.

Example: $17 \equiv 3$ *modulo* 7 and $-3 \equiv -13$ *modulo* 5.

2.7 Fundamental Rules of Counting

Fundamental Rules of Counting

In our daily life, we make use of mathematics all the time, both consciously and unconsciously.

Following a simple recipe to bake a cake for your daughter on her birthday can be one of your worst days if you don't do some mathematics that will involve counting the right amount of eggs to use and measuring the correct volume in liter or deciliter of milk, water and flour and so on ...

We might even ask ourselves questions like:
in how many ways can one distribute six different flavors of chocolate to three kids?

These type of questions require that we make use of some mathematical tools so that we can address them effectively . Let's introduce some basic principles of counting.

The Rule of Sum

Definition: If event A can be done in m different ways, while a second event B can be done in n different ways and the two events cannot be done at the same

time, then either event A or event B can occur in $m + n$ ways.

Example:

A local bookstore has 10 books on cryptography and 20 books on cryptanalysis. In how many ways can you select among those books to learn more about either cryptography or cryptanalysis?

Solution:

By the rule of sum, you can select among $10 + 20 = 30$ books in order to learn more about either cryptography or cryptanalysis.

The Rule of Product

Definition: If event A can occur in m different ways while event B can occur in n different ways, then there are $m \times n$ possible ways for both events to occur.

Example:

You have just won three roundtrip tickets at the National Lottery to three preselected different cities denoted by A, B and C. In how many ways can you visit each of these cities?

Solution:

We write ABC when the city A is visited first, then B and finally C. The number of ways can be calculated as follows: $3 \times 2 \times 1 = 6$. We see that this example involves the product of consecutive positive integers. This brings us to another rule known as the Factorial Rule.

The Factorial Rule

Definition: For n different items, there are $n!$ (pronounced n factorial) arrangements. More specifically, for an integer $n \geq 0$, n factorial is defined by:

$$n! = (n) \times (n-1) \times (n-2) \cdots 3 \times 2 \times 1, \qquad for\ n \geq 1.$$

However, we note that 0! (pronounced zero factorial) is given by:

$0! = 1$

Example

Suppose you need to arrange in a row and from left to right five distinct objects colored with red, green, black, yellow and white on a table. How many options do you have?

Solution

Before making any choice, you have five objects to choose from, If you place the black object first, then you still have four objects to choose from, namely the red, the green, the white and the yellow object.
Next, if you pick up the red object, you are left with three objects to choose from and so on until you place the last object.
Obviously, the number of ways to choose from these five objects can be calculated as: $5 \times 4 \times 3 \times 2 \times 1 = 5! = 120$. In this type of arrangement all the items are arranged. What if we need only arrange some items from a given set? This is where permutation comes in. It represents any linear arrangement of some objects of interest. We define perutation in the next section.

2.8 Permutation, Combination and Probability

Permutation

Definition: $P(n, r) = \frac{n!}{(n-r)!}$, with $1 \leq r \leq n$ and where r is the number of items to arrange from a collection of n items.

Example

In how many ways can you arrange (or choose) two persons from a set of five?

Solution

$P(5, 2) = \frac{5!}{(5-2)!} = \frac{5!}{3!} = 5 \times 4 = 20.$

Another way to solve this problem is to realize that for the first choice we have five options, and for the second choice we have 4 options which gives us, by the rule of product, $5 \times 4 = 20$.

Note that in this type of arrangements the position taken by each items in the arrangement matters. We usually assume the items are distinct and replacement is not allowed, that is, you cannot put back an item before the next choice.

Permutation with replacement

Sometimes it is not possible to distinguish some items from a given collection. Consequently we no longer have to deal with a collection of distinct objects, but a collection where some objects are identical.

Example

In how many ways can you arrange the six letters in the word ACCESS?

Solution

If we distinguish to two C's and the two S's as C1, C2, S1 and S2, then we can apply our prior knowledge on permutation of distinct objects to calculate that we have $6! = 720$ permutations.
However, in the event that the two C's and S's are by no means distinguishable; we need to take care such that the same arrangement is not counted twice. The letters that repeat are C and S. The C's can be arranged in $2! = 2$ ways, and the S's can also be arranged in $2! = 2$ ways. Thus the number of arrangements is $\frac{6!}{(2!\times 2!)} = \frac{6\times 5\times 4\times 3\times 2\times 1}{4} = 180$.

Circular Permutation

Sometimes we need to make circular arrangements instead of linear arrangements where items are to be arranged around a circle.

Example

Four people are seated around a circular table, how many different arrangements are possible? Note that arrangements are considered identical if one can be obtained from the other by rotation.

In the figure 3, all arrangements are considered identical because any one can be obtained from any other by rotation. Then by starting from the upper left corner and moving clockwise we are able to list the following distinct linear arrangements:

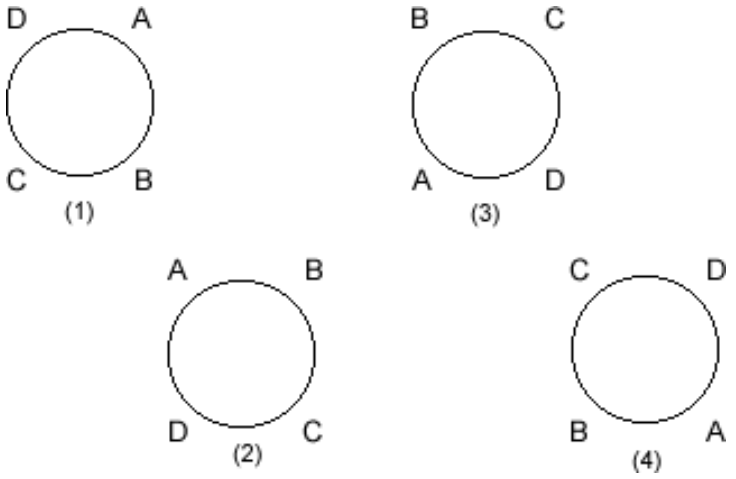


Figure 3: *Circular permutation.*

(1) gives DABC
(2) gives ABCD
(3) gives BCDA
(4) gives CDAB

which are all the same circular arrangement. We see that each circular arrangement corresponds to four linear arrangements, so we can conclude that 4 x (number of circular arrangements) = (number of linear arrangements).
We know that the number of linear arrangements of n distinct objects is $n!$, so in this particular case the number of linear arrangements = 4! Hence, the number of circular arrangements = $\frac{4!}{4} = \frac{4\times 3!}{4} = 3! = 6$.

In general, the number of circular permutations of n distinct objects is $(n-1)!$

Combination

Definition: $C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!}$, with $0 \leq r \leq n$. Sometimes, the symbol $\binom{n}{r}$ is used as a replacement of $C(n,r)$; they are both read as "*n choose r*".

Example

You have a group of ten students in an Art class. In how many ways can you select two of them to represent the school in the National Art Quiz Contest?

Solution

Here, it is obvious that the order is irrelevant, so we can choose two students from a group of ten in $C(10,2) = \frac{10!}{2!(10-2)!} = \frac{10\times 9}{2\times 1} = 45$ ways.
Combination is basically an arrangement of object where the order or position of objects is irrelevant.

It helps to remember that whenever we deal with a counting problem of these kinds, the order and position of objects is a relevant piece of information that will enable us to solve the problem at hand. If order is relevant, then we need to approach the problem in terms of permutations and arrangement $P(n,r)$. If order is not relevant, then the problem is likely to be solved if we think of it in terms of combination $C(n,r)$.

Probability

Probability is the branch of mathematics that studies the likelihood of a given event to occur.

Example

A spinner has 8 equal sectors alternatively colored black or white and numbered from 1 to 8. What are the chances of landing on the number 7 after spinning the spinner? What are the chances of landing on a black sector?

Figure 4: *A spinner*

Solution

The chances of landing on the number 7 are 1 in 8, or one eighth. The chances of landing on a black sector are 4 in 8, or one half.

We define some common terms used in probability.

Experiment: the process of performing one or more trial(s) that can enable us to measure the possibility of one or more event(s) to occur.

Trial: a unit of an experiment. It is a process by which one outcome is observed.

Event: It is one or more outcomes of an experiment that may or may not occur.

Outcome: It is the result of a single trial of an experiment.

Probability: it is the measure of how likely an event is to occur. And it is frequently expressed on a scale from 0 (impossibility) to 1 (certainty). If an event is impossible to occur, its probability is 0. However, if an event is certain to occur, its probability is 1, all other events take on a value between 0 and 1. The following figure depicts this situation.

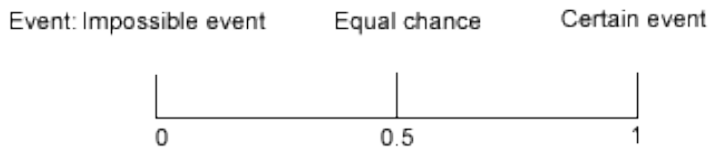


Figure 5: *Probability scale.*

The probability of event A is the number of outcomes involving event A divided by the total number of possible outcomes.

Example

A glass jar contains 14 red, 11 green, 13 blue and 16 yellow candies. If a single candy is chosen at random from the jar, what is the probability of choosing a red? a green? a blue? or a yellow candy?

Solution

In total, we have 54 candies in the jar, and this represents the total number of possible outcomes. The possible outcomes of this experiment are choosing a red, green, blue or a yellow candy. Hence we have:

$$P(choosing\ a\ red\ candy) = \frac{number\ of\ ways\ to\ choose\ a\ red\ candy}{total\ number\ of\ candies\ in\ the\ jar} = \frac{14}{54} = \frac{7}{27}$$

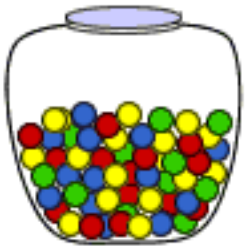


Figure 6: *A jar of candies.*

$$P(\textit{choosing a green candy}) = \frac{\textit{number of ways to choose a green candy}}{\textit{total number of candies in the jar}} = \frac{11}{54}$$

$$P(\textit{choosing a blue candy}) = \frac{\textit{number of ways to choose a blue candy}}{\textit{total number of candies in the jar}} = \frac{13}{54}$$

$$P(\textit{choosing a yellow candy}) = \frac{\textit{number of ways to choose a yellow candy}}{\textit{total number of candies in the jar}} = \frac{16}{54} = \frac{8}{27}$$

We can easily see that the outcomes in this experiment do not have the same chance to occur. We are more likely to choose a yellow candy than any other color. And we are least likely to choose a green candy. These assertions are proven mathematically. In fact the probability of each event enables us to say, for example, that we are more likely to choose a yellow candy from the jar than we are to choose a green candy.

2.9 Number Systems

Numbers like 1, 2, 3 and 4 are commodities to almost all societies today. They are used everyday and almost everywhere. They are used for counting, performing different calcutations or simply for representing values. A number system is a set of all symbols used to express quantities.

The Decimal system

The word decimal means ten, the decimal system is a system based on the ten arabic symbols or decimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

Example

The number 32 means three tens plus two (or three times ten plus two):

$32 = (3 \times 10) + 2$

The number 1976 means one thousand, nine hundreds, 7 tens, plus 6.

$1976 = (1 \times 1000) + (9 \times 100) + (7 \times 10) + 6$

The decimal system is said to have a base of 10. This means that every digit in a decimal number is mulitiplied by 10 raised to the power n, where n is the position occupied by that digit in the number.

Example

$32 = (3 \times 10^1) + (2 \times 10^0)$
$1976 = (1 \times 10^3) + (9 \times 10^2) + (7 \times 10^1) + (6 \times 10^0)$

The position occupied by the digits of a non-fractional number is determined from right to left starting at position 0 which is occupied by the rightmost digit.

$X = (\ldots x_3 x_2 x_1 x_0)$

This way of reading from right to left reminds us of the arabic standard, but then we recall that the symbols themselves are arabic, so this probably explains why, and it could not have been done otherwise without complicating something simple.

Fractional numbers are represented in a similar way.

Example

$32.76 = (3 \times 10^1) + (2 \times 10^0) + (7 \times 10^{-1}) + (6 \times 10^{-2})$

Generally, decimal numbers are represented as $X = (\ldots x_3 x_2 x_1 x_0 x_{-1} x_{-2} x_{-3} \ldots)$, and its value can be written in the following general form:

$$\sum_{i \in \mathbb{Z}}^{n} x_i 10^i$$

where n is the position of the leftmost digit and $i \in \mathbb{Z}$ (the set of all integers = ...-2, -3, -1, 0, 1, 2, 3 ...). We note that the position of digits in a non-fractional numbers starts at x_0 and moves to the left, thus position $x_{-1} x_{-2} \ldots$ are simply irrelevant when dealing with non-fractional numbers.

The Binary system

We have seen that ten digits are used to represent numbers in the decimal system. Therefore the decimal system is said to have a base or radix of 10. Unlike the

decimal system, the binary system only uses two digits, 0 and 1, to represent all numbers. Thus, the binary system is said to have a base or radix of 2. This also means that each digit in a binary number is multiplied by 2 raised to the power n, where n is the position occupied by that digit in the binary number. To avoid confusion, let's we put a subscript on the number to clearly indicate its base. Like 32_{10}, and 76_{10} are decimal numbers (base 10), wheras 100_2 and 101_2 are binary numbers.

Example

The following means that the binary number 10 equals the decimal number 2.

$10_2 = (1 \times 2^1) + (0 \times 2^0) = 2_{10}$

The next equality means the binary number 101 equals the decimal number 5.

$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 5_{10}$

Fractional binary numbers are also represented with negative powers of 2.

$101.101_2 = (1 \times 2^2) + (0 \times 2^2) + (1 \times 2^2) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) = 5.625_{10}$.

Hexadecimal system

This system uses sixteen symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) to represent numbers. These symbols are the hexadecimal digits. Thus, the hexadecimal system is usually said to have a base or radix of 16. Recall that hash values are represented by using hexadecimal digits. This notation is a more compact and human-friendly way of representing binary numbers. Binary numbers are grouped into sets of four digits where each possible combination of four binary digits is assigned to one hexadecimal symbol.

$0000_2 = 0_{16}$	$1000_2 = 8_{16}$
$0001_2 = 1_{16}$	$1001_2 = 9_{16}$
$0010_2 = 2_{16}$	$1010_2 = A_{16}$
$0011_2 = 3_{16}$	$1011_2 = B_{16}$
$0100_2 = 4_{16}$	$1100_2 = C_{16}$
$0101_2 = 5_{16}$	$1101_2 = D_{16}$
$0110_2 = 6_{16}$	$1110_2 = E_{16}$
$0111_2 = 7_{16}$	$1111_2 = F_{16}$

Hexadecimal notation can also be used to represent decimal numbers. And the conversion from hexadecimal to binary is done as follows:

$$\begin{aligned} 3D_{16} &= (3_{16} \times 16^1) + (D_{16} \times 16^0) \\ &= (3_{10} \times 16^1) + (13_{10} \times 16^0) \\ &= 48_{10} + 13_{10} = 61_{10} \end{aligned}$$

On the other hand, converting from hexadecimal to binary is as simple as replacing each hexadecimal digit with its binary equivalent.

Example

$$\begin{aligned} 0011_2 &= 3_{16} \\ 1101_2 &= D_{16} \\ 00111101_2 &= 3D_{16} \end{aligned}$$

We can appreciate the ease of conversion between these two number systems. Hexadecimal characters are sometimes used to represent a single byte (8 bits) in the computing environment where each byte is represented as two hexadecimal characters.

Example

$10011001_2 = 128 + 16 + 8 + 1 = 153_{10} = 99_{16}$

Hexadecimal symbols are also used to encode colors in HTML (Hyper Text Markup Language) which is the language used to create web page. An HTML file is a text file containing markup tags which tell the web browser how to display a web page.

2.10 Hash Function, Birthday Paradox

Hashing

Hashing is a process by which one turns a string of characters with variable length into a fixed-length value which represents the original string.

Hashing versus Encrypting

Sometimes, hashing is being referred to in situations where encryption is the most appropriate term and vice versa. We clarify this common confusion once and for all.

Hashing ,in cryptography, is a one-way operation which transforms a stream of data into a more compressed form called a message digest. The operation is not be invertible, meaning that recovering the original data stream from the message digest should not be possible. All the message digests or hash values generated by a given hash function have the same size no matter what the size of the input value is.

Encryption on the other hand, can be thought of as a two-way operation which transforms a plaintext into a ciphertext and allows for the process to be inverted by transforming the ciphertext back into its original plaintext via a mechanism called decryption. Both operations depend on a key.

Encrypting versus Encoding

Encoding and decoding are sometimes used to describe encryption and decryption respectively. But can we really substitute encoding for encryption without creating any confusion in most people's mind?

Encryption, as mentioned above, is a process that transforms information from its original, and usually comprehensible, form into a more disguised and unintelligible form. The opposite process of recovering the original message from its disguised form is called decryption. The driving force behind encryption is to keep a piece of information secret to all but those authorized to read that information. You know you are authorized to read an encrypted message if you possess the key that will allow you to decrypt the ciphertext back to its original form called plaintext. Thus, the main goal of encryption is to keep data secret by concealing its content.

Encoding, which is sometimes used and accepted as a synonym for encryption, is more directed at converting some data to a format that will facilitate its efficient manipulation, transmission and storage in the digital world. Encoding does not conceal the content of data; it only converts the data to a format that can be efficiently managed by our electronic devices (computer, mobile phone, television etc), transmission media (cables and wires), storage devices (hard disk, pen drive), and applications software (web browser, mail client etc).

There are many encoding techniques used to convert data to different format depending of what we wish to achieve. We list some of them as an example: Character encoding is a method of converting letters, numbers and other symbols into integers and 7-bit (a string of 7 0s or 1s) or 8-bit binary versions of those integers. The ASCII (American Standard Code for Information Interchange) character set is the most common encoding format for text files in computer.

Cryptography

Cryptography is one of the two branches that make up *cryptology*; the other branch, *cryptanalysis* attempts to undo what cryptography tries to do.

Cryptography is the science that aims at designing and developing cryptographic systems, sometimes referred to as a *cryptosystems*. A cryptosystem is a set of methods needed to create a particular encryption and decryption scheme. A typical cryptosystem is made up of three parts: One that generates the encryption/decryption key, one that performs the encryption process, and one that deals with the decryption process.

Encryption is the process by which one changes a message (called plaintext) in order to render it unreadable to all but those possessing the decryption key. The unreadable message is usually referred to as the ciphertext.

Decryption is the inverse process which recovers the plaintext from the ciphertext.

Cryptographic Hash Functions

A hash function, is a function that takes some message of any length as input and transforms it into a fixed-length output called a *hash value*, a *message digest*, a *checksum*, or a *digital fingerprint*.

A hash function is a function $f : D \rightarrow R$, where the domain $D = \{0,1\}^*$, which means that the elements of the domain consist of binary string of variable length; and the range $R = \{0,1\}^n$ for some $n \geq 1$, which means that the elements of the range are binary string of fixed-length. So, f is a function which takes as input a message M of any size and produces a fixed-length hash result h of size n. A hash function f is referred to as compression function when its domain D is *finite*, in other word, when the function f takes as input a fixed-length message and produces a shorter fixed-length output.

A cryptographic hash function H is a hash function with additional security properties:

1. H should accept a block of data of any size as input.
2. H should produce a fixed-length output no matter what the length of the

input data is.

3. H should behave like random function while being deterministic and efficiently reproducible. H should accept an input of any length, and outputs a random string of fixed length. H should be deterministic and efficiently reproducible in that whenever the same input is given, H should always produce the same output.

4. Given a message M, it is easy to compute its corresponding digest h; meaning that h can be computed in polynomial time $\mathrm{O}(n)$ where n is the length of the input message, this makes hardware and software implementations cheap and practical.

5. Given a message digest h, it is computationally difficult to find M such that $H(M) = h$. This is called the one-way or pre-image resistance property. It simply means that one should not be capable of recovering the original message from its hash value.

6. Given a message M_1, it is computationally infeasible to find another message $M_2 \neq M_1$ with $H(M_1) = H(M_2)$. This is called the weak collision resistance or second preimage resistance property.

7. It is computationally infeasible to find any pair of distinct messages (M_1, M_2) such that $H(M_1) = H(M_2)$. This is referred to as the strong collision resistance property.

Remarks: property 7 implies both property 5 and 6.

These properties are required in order to prevent or withstand certain types of attacks which may render a cryptographic hash function useless and insecure. In addition to producing a "digital fingerprint" of a message M that is unique and to providing strong collision resistance, a cryptographic hash function should also be highly sensitive to the smallest change in the input message. Such that a change, as small as a single digit, in the input message should produce a large change in the hash value of the message. Note that a message in this context can be a binary text file, audio file, or executable program.

The security of the hash function does not originate in keeping the hash function itself secret but comes from its ability to produce one-way hash values alongside with the property of being collision-free, we talk about collision when two or more different messages results in the exact same hash value. So far we have talked about hash functions used without a key, but hash functions can be used with a key; both symmetric (shared key) and asymmetric keys can be used; in which case

the function is called a message authentication code or MAC.

The strong collision resistance mentioned above is a necessary security property that results from a method of finding collision known as the birthday attack based on the birthday paradox.

The birthday paradox, which we will explain in great detail shortly, states that in a group of 23 people chosen randomly, the probability that two of them share the same birthday is at least $\frac{1}{2}$. In fact, if the attacker is not able to find a faster way to come up with a pair of preimages $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$, then he or she will have to collect a large amount of messages M_i and their corresponding hash values $H(M_i)$, sort them and look for a match, this is known as a brute-force search attack. If the size of the hash values is n-bit, then there are 2^n possible hash values and the attacker will have to compute about the square root of this value, namely $2^{\frac{n}{2}}$, before he or she can expect to find a match. To understand this fact, we need to look closer at both the birthday paradox and the birthday attack.

The Birthday Paradox

The birthday paradox is a fascinating problem which demonstrates the nonintuitive character of probability results. The problem can be stated as follows:

Assume that birthdays are evenly distributed throughout the year and ignore leap years when February has 29 days, if we have a room with k random people in it, what is the minimum value of k such that the chances that two of them have the same birthday is greater than or equal to 0.5?

We know that a non-leap year consists of 365 days, some of us might reasonably argue that the chances will not reach 100% until there are 366 people in that room. The birthday paradox tells us that things are not always as simple as they appear at first glance, and that you can have far less than 366 people in that room and still achieve the same probability! Let's go deeper into the paradox to understand it.

Let $D(n, r) = Pr(\textit{there is at least one duplicate in } r \textit{ birthdays})$, with $1 \leq r \leq n$ and where each birthday is equally likely to take on a value between 1 and n.

In our case, $n = 365$ and we are trying to find the smallest r which gives us $D(n, r) \geq 0.5$. We define $Q(365, r)$ to be the probability that there are no duplicates in r birthdays. With the r birthdays representing the number of people in the group under consideration.

We see that if $r > 365$, then it is not possible for all birthdays to be different. We note that r people can be paired up in $\frac{r!}{2!(r-2)!}$ ways, so 30 people can be paired up in 435 ways. Thus, we assume that $r \leq 365$. The number of different ways that we can have r birthdays with no duplicates is similar to the number of different ways that we can arrange r objects from a set of 365 objects where the position and order is relevant. The solution is given by the rule of permutation, as for the first birthday we have 365 choices, for the second we have 364 choices and so on. Hence, the number of different ways to have r birthdays with no duplicates is

$365 \times 364 \times \cdots \times (365 - r + 1) = \frac{n!}{(n-r)!} = \frac{365!}{(365-r)!}$.

If duplicate birthdays are allowed, then for the first birthday we have 365 choices, for the second we have 365 choices and so on. And the total number of possible birthdays is

$365 \times 365 \times \cdots 365 \quad (r\ times) = 365^r$.

The probability that there are no duplicates in r birthdays is equal to the number of different ways that we can have r birthdays with no duplicates divided by the total number of possible birthdays:

$Q(365, r) = \frac{\frac{365!}{(365-r)!}}{365^r} = \frac{365!}{(365-r)!365^r}$.

The probability that there is at least one duplicate in r birthdays is equal to:

1 - the probability that there are no duplicates in r birthdays.

$D(n, r) = 1 - Q(365, r) = 1 - \frac{365!}{(365-r)!365^r}$.

We find that for $r = 23, D(365, 23) = 0.5073$, which is greater than or equal to 0.5.

Hence, in a group of 23 randomly chosen people, there is fifty-fifty chance that two of them share the same birthday. And for $r = 100, D(365, 100) = 0.9999997$, meaning that in group of hundred people it is almost certain that at least two of them have the same birthday.

Before we jump into any conclusion, let's remind that those probabilities hold true based on the assumptions made earlier, namely that birthdays are evenly distributed throughout the year. In reality, birthdays are not distributed perfectly throughout the year, so depending on the people and the way their birthday is distributed in a year, these probabilities might vary accordingly.

The Birthday Attack

The birthday attack is a class of brute-force technique which exploits the idea behind the birthday paradox to solve the problem of finding collision in some class of cryptographic hash function faster than by brute-force search attack. Let's look at a similar problem first.

Given a cryptographic hash function H, with n possible outputs and a known hash value $H(x)$, If H is applied to k random inputs, then what is the smallest k such that the probability of having at least one input y from the set k satisfying $H(y) = H(x)$ is 0.5?

If $k = 1$, then the probability of having at least one input y from k such that $H(y) = H(x) = \frac{1}{n}$. Conversely, the probability of having at least one input y from k such that $H(y) \neq H(x) = 1 - \frac{1}{n}$.

If $k > 1$ random inputs are generated, then the chances that none of them satisfies $H(y) = H(x)$ is equal to the product of the probability that each of them satisfies $H(y) \neq H(x)$ and it is equal to:

$(1 - \frac{1}{n})(1 - \frac{1}{n}) \cdots (1 - \frac{1}{n}) \quad (k\ times) \quad \Leftrightarrow \quad (1 - \frac{1}{n})^k.$

Hence, the probability that there is at least one match is

$1 - Pr(the\ probability\ that\ there\ is\ no\ match)$, and it can be written as $1-(1-\frac{1}{n})^k$

We recall the binomial theorem which states what follows:

$(1 - a)^k = 1 - ka + \frac{k(k-1)}{2!}a^2 - \frac{k(k-1)(k-2)}{3!}a^3 \cdots$

When a is very small, this equality can be approximated to $(1 - ka)$. Going back to our problem, we find that the probability of having at least one match can be approximated to:

$1 - (1 - \frac{1}{n})^k \approx 1 - (1 - \frac{k}{n})$

If the probability is 0.5, we find that $\frac{k}{n} = \frac{1}{2} \Leftrightarrow k = \frac{n}{2}$

For a hash function with n possible outputs, it is enough to generate the hash value of $\frac{n}{2}$ inputs in order to expect to have a match with a probability of 0.5.

This problem can be generalized to the problem of finding the minimum values of k such that there is at least one duplicate. The following inequality statement will help us in the generalization of the birthday problem:

$(1 - x) \leq e^{-x} \qquad \forall x \in \mathbb{R}$

Proof:

we consider the function $f(x) = e^x - (1 + x)$ and find its derivative to be $f'(x) =$

$e^x - 1$. Next, we find the minimum of its derivative $f'(x)$. We note that $f'(x) = 0$ gives $x = 0$. We see that for $x \geq 0$, $f'(x) \geq 0$, and for $x < 0$, $f'(x) < 0$, therefore $x = 0$ gives the minimum for the derivative $f'(x)$ of $f(x)$. Hence, $f(x) \geq 0$.

We note that for small values of x, $(1 - x) \approx e^{-x}$, meaning that $(1 - x)$ can be approximated to e^{-x} , as shown by figure 7 which represents the graph of both function $g(x) = (1 - x)$ and $f(x) = e^{-x}$.

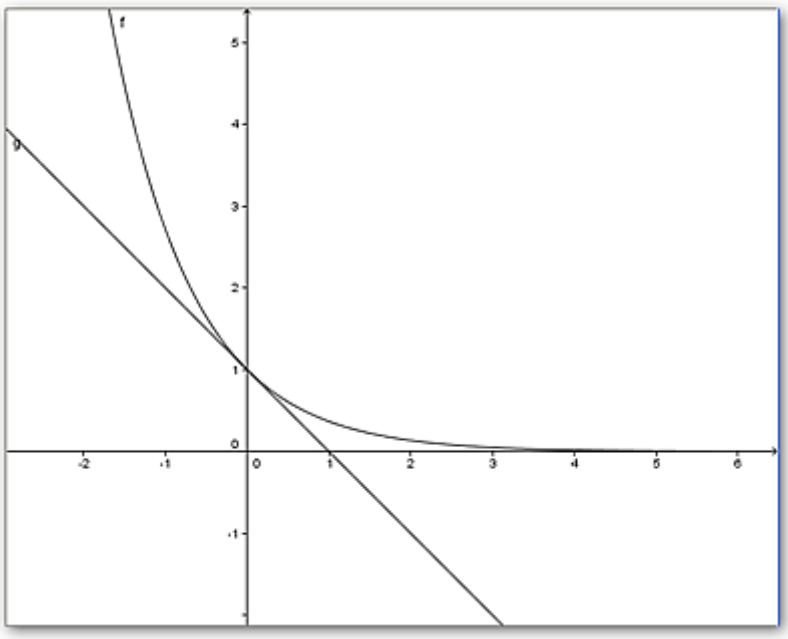


Figure 7: *Graph of functions* $f(x) = e^{-x}$ *and* $g(x) = 1 - x$

Now, we restate the initial birthday problem to a more general one as follows:

Suppose we have a hash function H, with 2^n possible outputs (the hash function produces an n-bit output). If H is applied to k random inputs, what is the smallest value of k which will ensure us that there is at least one duplicate?

We recall that the probability of having at least one duplicate in r birthdays where each birthday is equally likely to take on a value between 1 and n is given by:

$D(n, r) = 1 - Q(n, r) = 1 - \frac{n!}{(n-r)!n^r}$

Following the same reasoning, we express the probability of having at least one duplicate in k random inputs where each input is equally likely to take on an

output value between 1 and 2^n as

$$
\begin{aligned}
D(n,k) &= 1 - Q(n,k) = 1 - \frac{n!}{(n-k)!n^k} \\
D(n,k) &= 1 - \frac{n(n-1)\cdots(n-k+1)}{n^k} \\
&= 1 - \left[\left(\frac{n}{n}\right)\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right)\right] \\
&= 1 - \left[\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right)\right] \\
&= 1 - \left[\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{k-1}{n}\right)\right]
\end{aligned}
$$

By making use of the above approximation pictorially represented in figure 7 we have:

$$
\begin{aligned}
D(n,k) &\approx 1 - \left[(e^{-\frac{1}{n}})(e^{-\frac{2}{n}})\cdots(e^{-\frac{k-1}{n}})\right] \\
&\approx 1 - e^{-\left[\frac{1}{n}+\frac{2}{n}+\cdots+\frac{k-1}{n}\right]} \\
&\approx 1 - e^{-\frac{k(k-1)}{2n}}
\end{aligned}
$$

We know from of the summation formulas that:

$$
S = \sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2}
$$

This can be proven by simply writing the sum forwards and backwards and adding the two to get

$S = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$

$S = n + (n-1) + (n-2) + \cdots + 3 + 2 + 1$

$2S = (n+1) + (n+1) + (n+1) + \cdots + (n+1) + (n+1) = n(n+1)$

$S = \frac{n(n+1)}{2}$,

therefore, we could write above $\frac{1}{n} + \frac{2}{n} + \cdots + \frac{k-1}{n} = \frac{k(k-1)}{2n}$.

If we are interesting in knowing the smallest value k for which the probability

$D(n,k) > 0.5$, then we just have to replace $D(n,k)$ by 0.5 in the equality that we just derived, namely,

$D(n,k) = 1 - e^{-\frac{k(k-1)}{2n}}$,

which gives

$$\begin{aligned} \frac{1}{2} &= 1 - e^{-\frac{k(k-1)}{2n}} \\ 2 &= e^{\frac{k(k-1)}{2n}} \\ \ln 2 &= \frac{k(k-1)}{2n} \end{aligned}$$

For very large value of k, we can approximate $k(k-1)$ to k^2, so the equation becomes:

$$\begin{aligned} \ln 2 &= \frac{k^2}{2n} \\ k^2 &= (2 \ln 2)n \\ k &= \sqrt{(2 \ln 2)n} = 1.18\sqrt{n} \end{aligned}$$

We check that this result holds by replacing n by 365 and find that $k = 1.18\sqrt{365} = 22.54 \approx 23$ which is the result we found in the section dealing with the birthday paradox.

We have shown that with the birthday attack, sometimes called the square root attack, we only need to apply $k = \sqrt{2^n} = 2^{\frac{n}{2}}$ random inputs to a hash function which produces 2^n outputs in order to expect to find a collision with the probability greater than or equal to 0.5! This is one of, if not mainly, the reason why the size of the hash value of modern hash functions is required to be large enough to make a birthday attack computationally infeasible.

Remarks

Birthday attack is impractical due to the fact that it requires a huge amount memory space, on the order of $2^{\frac{n}{2}}$ for a hash value size of n. The same running time with polynomial in n memory requirement is achieved with Floyd's cycle-finding algorithm and improvements to it (see [28], chapter 9, p.369-370).

Part III
Application of hash function in cryptography

3 Application of hash function in cryptography

Hash functions are used in many situations to support various cryptographic protocols. Their most common application is the creation and verification of digital signature (a means to verify the authenticity of an electronic document).

3.1 Digital Signature

We all know what a hand-written signature is and we certainly understand its purpose. It is a way to prove that a paper document is signed by us and not by someone else. To prove this, the current hand-written signature is compared with one or more of our previous hand-written signatures. If there is a match then the recipient of the document can safely accept that the document could not have been signed by someone else. In case it is the first time, we have to prove our identity by means of some identification card, and necessarily by being physically present to sign the document.

Some properties of the hand-written signatures are:

- ⋆ *The signature should be unique to each person.*
- ⋆ *The signature should be verifiable as belonging a particular person.*

The digital signature is the electronic equivalent to the hand-written signature with regard to its purpose. More precisely, a digital signature is a sort electronic "stamp" or digital "fingerprint" placed on a document that is unique to the signer of the document and to the signed document. One major difference between a digital and a hand-written signature is that for every different document, the digital signature is different even if the signer and the private/public key pair are the same. We also note that a digital signature scheme provides both data integrity protection and data origin authentication.

The application of the hash function in a digital signature scheme works as follows: We consider Sarah as both the sender and the signer of the document.

Sarah holds a private/public key pair, the hash function used to create the message digest of the document and the document itself (see figure below).

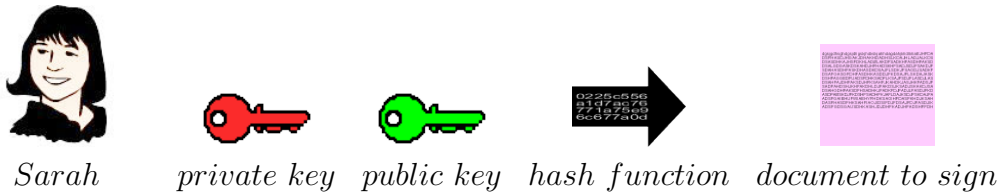


Figure 8: *Sarah, the signer and sender of the document.*

And we take Remy as the recipient of the signed document.
To digitally sign a document, Sarah generates the hash value of the message or document she wishes to transmit to Remy.

Figure 9: *Applying the hash function to the document to generate its message digest.*

Next, Sarah encrypts with her private key the message digest to produce the signature.

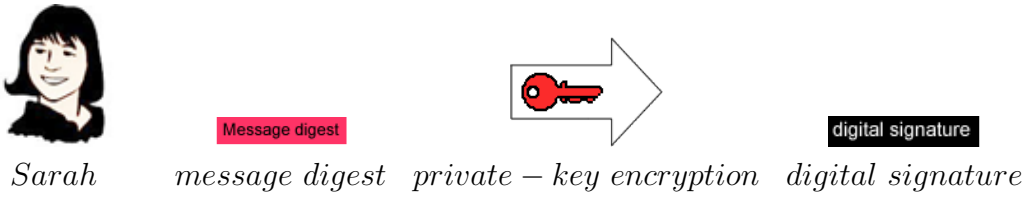


Figure 10: *Applying the private-key encryption to message digest to generate digital signature.*

Now, Sarah appends the digital signature to the document.

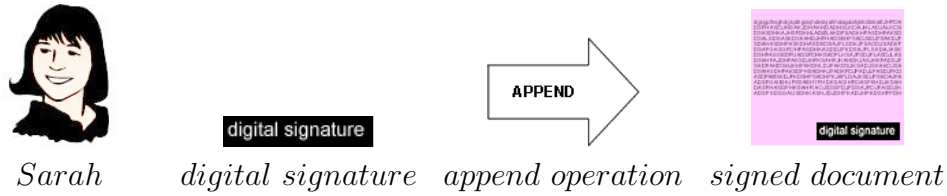


Figure 11: *Appending digital signature to document.*

Finally, Sarah encrypts the signed document with her private key and transmits it to Remy.

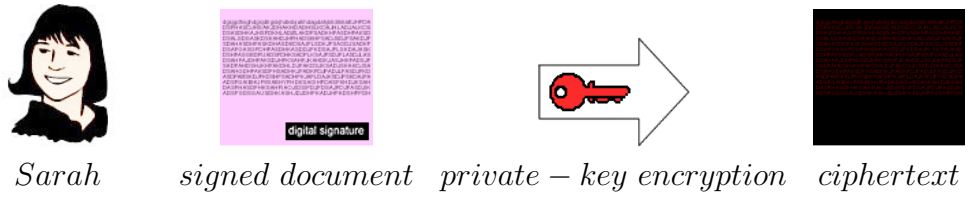


Figure 12: *Encryption of the signed document to generate the ciphertext.*

When Remy receives the ciphertext, He first decrypts the ciphertext using Sarah's public key to obtain the original signed document.

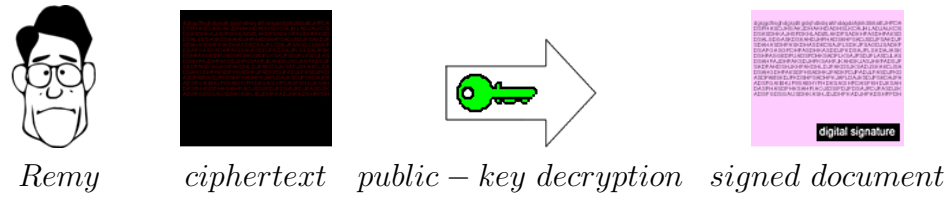


Figure 13: *Decryption of the ciphertext to generate the signed document.*

Then Remy "splits" the signature and the document. This is not an operation in its own right. The signature is not really physically glued to the signed document. And by decrypting the ciphertext, the recipient gets the document and the signature separated from each other and can simply read one or the other. So

the split operation can simply be understood as an operation where the recipient distinguishes unambiguously the signature from the signed document.

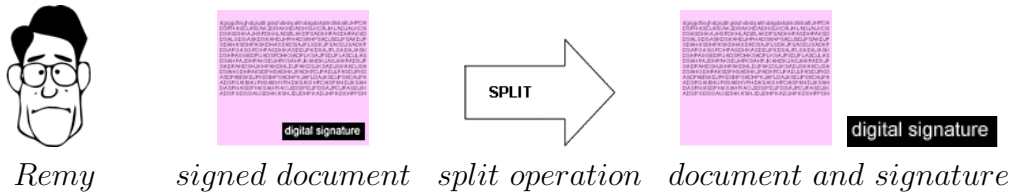


Figure 14: *Distinguishing document from it's signature.*

Next, Remy generates a new message digest of the received document using the same hash function used by Sarah.

Figure 15: *Generating a new message digest from received document.*

Remy may concurrently decrypt the digital signature using Sarah's public key to obtain the original message digest of the document previously generated by Sarah.

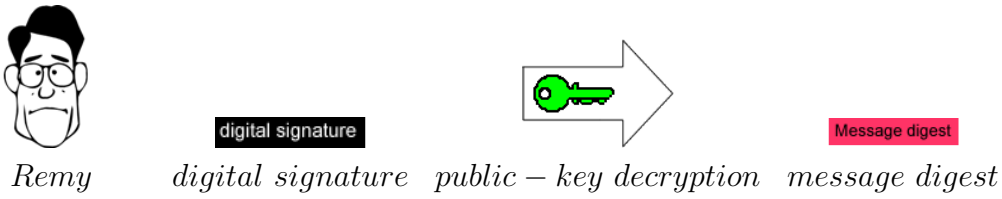


Figure 16: *Decryption using public key.*

What Remy finally does, is to compare the new message digest he just generated with the message digest transmitted by Sarah. If there is a match, then Remy can be assured that the document has been signed by Sarah and that the document

has not been altered along the way. If there is no perfect match, then authenticity of neither the document nor the signer nor the signature can be verified.

However, we note in this particular case that if a third person is able to maliciously intercept the ciphertext and modify both the message and its corresponding hash value before it gets to Remy, then Remy will have no way of knowing that the document has been altered in transit.

3.2 File Integrity Verification

Hash functions are widely used to verify file integrity. And in the paragraph on digital signature above, it is clear that the message digest is used to verify the integrity of the document. Indeed, it certifies that the document has not been modified somewhere between the moment it was sent and the moment it was received. Those who have once downloaded free software on the Internet, have probably visited websites which publish the checksum of the software near the hyperlink of the binary executable file or the archived source code of the corresponding software. Without this vital piece of information which is the checksum of the software, one will have a hard time verifying the integrity of downloaded software.

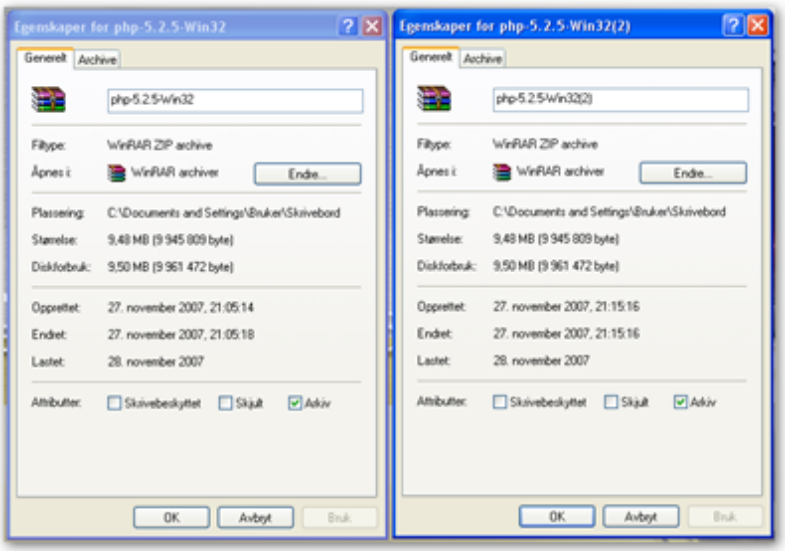


Figure 17: *File comparison by size.*

Let's picture an attempt at verifying a file downloaded from the Internet without the use of its checksum. We download the file twice and compare the bits or the size of the downloaded files. If they are the same, then we are probably good to go. Figure 17 depicts a situation where one can presume that the downloaded file is authentic.

But if they differ by size, then we have a problem. Which one is authentic? To answer this, we need yet another download and comparison. If all three files are different from one another, then the situation gets scary if not frustrating. And, we might quite easily end up with a bunch of files with no clear way of determining the authentic one. Well, life gets a lot easier when the checksum is published along with a hyperlink that one can click to download the corresponding file.

We consider the PHP (Personal Home Page) project located at the following Internet address, www.php.net. PHP is a free cross-platform server-side scripting language for web development which can be embedded into HTML (Hyper Text Markup Language) files to dynamically create web pages among other things. At the download page of the project, the md5 (Message Digest 5 algorithm) checksum of all downloadable files is unambiguously published under the hyperlink of the corresponding source code or binary executable file (see figure 18).

Figure 18: *PHP source code and binary file width corresponding md5 checksum.*

All we need to do in order to verify the integrity of the file is to generate the

md5sum of the file we just downloaded and compare it with the one published at the project's website.

Figure 19: *Checksum comparison: There is a perfect match.*

Note that the two checksums have to match exactly. If the checksums differ by only one bit or character then the two files are not the same! There is no approximation when it comes to cryptographic hashes. Either both inputs are identical and their hash values match perfectly or both inputs differ and so do their respective hash value.

3.3 Password Hashing

A password, in computer science, is a secret sequence of character that one uses to gain access to a file, an application or a computer system. Password has been used long before our time. It used to be a secret word or phrase which enabled a person to be accepted as a friend by soldiers posted to keep watch and guard. In our modern and more computerized world, it is a secret data that one has to input to a computer system in order to be granted access to the resources of that system.

Password hashing was used since the early ages of the UNIX operating system. Users of UNIX systems have their password hashed and stored in a password file. Today, many web applications use a database to store and retrieve a variety of data including passwords. A poor practice is to store passwords in cleartext (the original form) wherever they are located in the computer system. If someone can somehow get to that location then the person will easily possess all passwords available there. Fortunately, some web applications generate a hash value of all passwords and store these hash values, rather than the password itself, in the database.

Let's consider a simple example which illustrates how password hashing works in practice. We consider the open source Content Management System project called Joomla! (www.joomla.org). Once installed, the application offers a backend section. This is where users can login by entering a username and password combination in order to gain access to the resources of the website via an administration console.

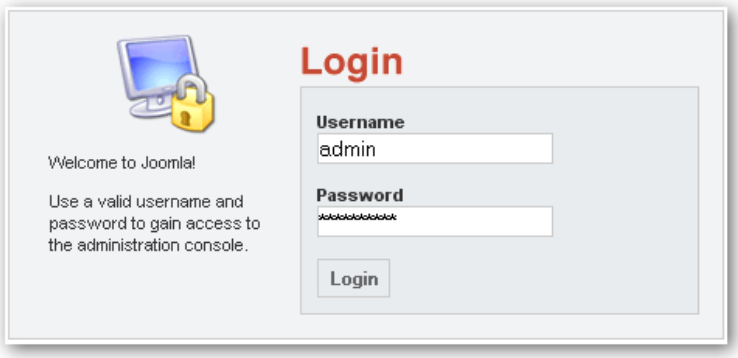


Figure 20: *Joomla back-end login window.*

All passwords in this application (up to version 1.0.12) are hashed with the md5 algorithm and the resulting hash value is stored in the database.

id	name	username	email	password	usertype
62	Administrator	admin	admin@email.com	11f953d5321942a7f01e4317d718bec	Super Administrator

Figure 21: *Hash of password stored in database.*

When a user enters his/her credential at the backend login page, the password entered in cleartext is first hashed with the md5 algorithm and the output is compared against the value of the hashed password stored in the database for which the usernames are identical. If the two strings match then access is granted, otherwise access is denied.

3.4 Key Derivation

Key derivation is the process of deriving various keys from a shared secret password or passphrase (which typically does not have the desired properties to be used directly as cryptographic keys) to secure a communication session. For example, two people can agree on a secret key and pass that key to a key derivation function to produce keys fro encryption and authentication. This guarantees that an attacker who learns your authentication key will not have access to your encryption key.

The key derivation function can be expressed as follows:

$DK = KDF(SecretKey, Salt, Iterations)$

where

DK is the derived key, KDF is the key derivation function, $SecretKey$ is the original shared secret (password or passphrase), $Salt$ is a random number which acts as cryptographic salt, and $Iterations$ refers to the number of iterations of a sub-function.

3.5 Trusted Digital Time-Stamping

Sometimes, it is desirable to bind a time together with a document as to certify its existence at that particular time. In the matter of intellectual property where dispute may arise between two or more people about who was the first one to make a discovery or an invention, time-stamping can play an important role at determining who is right. Let's take a closer look at what this is about.

A digital time-stamp is sort of digital "stamp" used to prove the existence of a digital document at a certain date. The creation date of digital documents can be modified and go undetected. The figure below illustrates an example a forward-dated document downloaded from the Internet on November 28th 2007. All we did was a change to the operating system's year to 2015 before we downloaded the file.

Thus, the creation date on a digital document is simply not reliable as a proof of the document's existence on the date that document claims to have been created. However, forward-dating is less attractive than back-dating for the simple fact that a "conscious" person will not believe that a document presented to him/her today was in fact created tomorrow!

To avoid date of creation "conflict", it is required that a Trusted Third Party (TTP) playing the role of a Time Stamping Authority (TSA) processes all valid

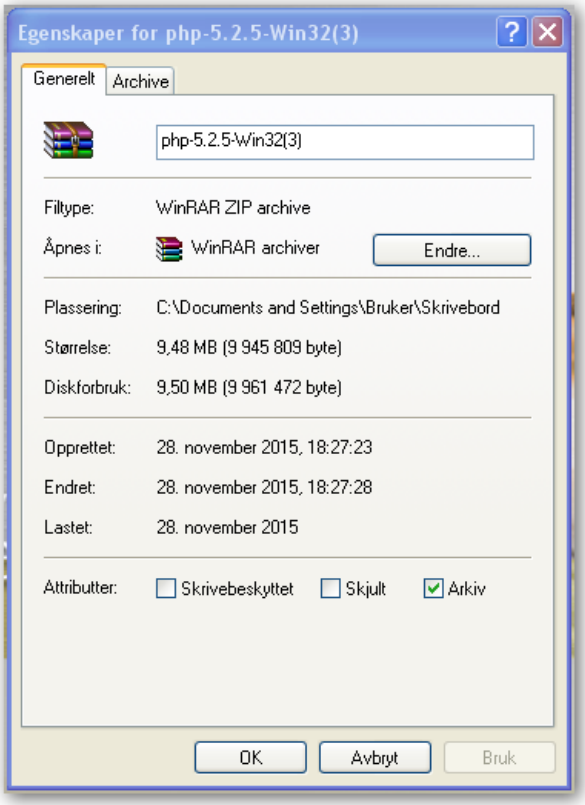


Figure 22: *A forward-dated document.*

digital time-stamping which can be used in a court of law in case of a copyright or patent dispute. Multiple time-stamping authorities can be contacted to increase the level of credibility of a document one wishes to time-stamp.

The process of creating a timestamp relies on digital signature scheme and hash functions, so does its security. This process basically involves two participants, the requesting entity (which is the person requesting a time-stamp token and initiating the entire process) and the Time Stamping Authority (which is the company that issues time-stamp tokens). The creation of a timestamp briefly occurs as follows:

- ⋆ The requesting entity calculates the hash of the document she/he wishes to have time-stamped and sends the resulting hash value as a request to the Time Stamping Authority.
- ⋆ The Time Stamping Authority appends a timestamp to the received hash value and calculates the hash of this concatenation. This final hash is digitally signed using the TSA's private key. Both the signature (the signed hash generated by the TSA) and the timestamp are sent as a response to the requesting entity.
- ⋆ Upon receipt of the response, the requesting entity should verify that the timestamp received matches perfectly with the timestamp requested. To verify this, the requesting entity decrypts the signed hash using the TSA's public key, let's call it TSA_HASH.
- ⋆ Next, the requesting entity concatenates the received timestamp to the exact same hash of the original document and calculates the hash of the result of this concatenation, let's call it OD_HASH. If TSA_HASH equals to OD_HASH then everything is alright, the timestamp is correct and was issued by the right Time Stamping Authority. The requesting entity may store all the data in a safe location.

If TSA_HASH is not equal to OD_HASH (and provided that the original document has not been modified since we sent the request) then either one of the following hypothesis holds true:

- ⋆ The timestamp was altered along the way.
- ⋆ We have received the wrong timestamp from the right TSA
- ⋆ We have received the wrong signature from the right TSA
- ⋆ The response was simply not issued by the right TSA

In any case, the TSA should immediately be notified of the situation.

One noticeable advantage of digital time-stamping is that the content of the original document is never revealed to the TSA. There are additional parameters included in a request sent to the TSA and in a response received from the TSA, but for the sake of simplicity there have not been mentioned here. For detailed information, please read the RFC (Request For Comment) 3161 which can be found at this Internet address http://tools.ietf.org/rfc/rfc3161.txt as of this writing.

3.6 Rootkit Detection

A Rootkit is a program or a set of programs that a hacker installs on the victim's computer in order to cover the tracks of other malicious programs which attempt to corrupt an operating system. A rootkit will hide its presence on a compromised system. It will replace or alter several legitimate system programs (such as "ls", "find", "locate", "top", "kill", "netstat" found on a UNIX system) by others which are specifically designed to prevent the rootkit's detection and removal. This means that once a rootkit is installed on a system, none of the programs on that system can be trusted to give precise information or to act as expected.

Rootkits can be detected in several ways including signature-based detection which uses scanning tools much like antivirus or antispyware programs that scan the system for signs of previously known rootkits patterns.

Another way of detecting the presence of rootkits includes behavior-based detection. For example, if the system's hard disk total size is 40 gigabytes and it has 10 gigabytes of files on it while it is reporting less than 30 GB of available free space. This behavior should raise suspicion about some files that are present on the system and which are not being reported by the system tool.

A third method of detecting the presence of rootkits which is more of interest to our subject involves the use of cryptographic hash functions and is called a $hash - based$ detection. With this method, a fingerprint or message digest of the filesystem or part of it is generated at regular intervals before and after any legitimate action which adds or removes files in the system. This fingerprint is later compared with the current state of the filesystem to find out if any unauthorized change has been made.

Suppose that as a system administrator of a web server, you decide to take fingerprint of a limited amount of carefully preselected "static" directories (by static we mean directories which do not house system log files that are being written to regularly by the system). Thus, you take a fingerprint or digest of those directories before leaving work every evening and when arriving at work every morning. You compare to check if the fingerprint taken the day before matches the one taken the day after. This routine is efficient at revealing any change made to the file system,

since the fingerprints won't match even if a change was made to a single bit in a file.

Part IV
Standards cryptographic hash functions

4 Standards cryptographic hash functions

Cryptographic hash functions come in different shape and size. There are basically two main categories of hash functions. Hash functions that depends on a key for their computation, usually known as Message Authentication Code or MAC and hash functions that do not depend on a key for their computation, generally known as un-keyed hash function or simply hash function.

All well known hash functions are either based on a block cipher or on modular arithmetic. But before stepping into their details, we study a well known method used to build the most popular hash functions, the Merkle-Damgård construction.

4.1 The Merkle-Damgård Construction

Named after its two inventors, the American Ralph C. Merkle and the Danish Ivan Damgård, the Merkle-Damgård structure defines a generic step by step procedure for deriving a fixed-length output value from a variable-length input value. The process is depicted in figure 23.

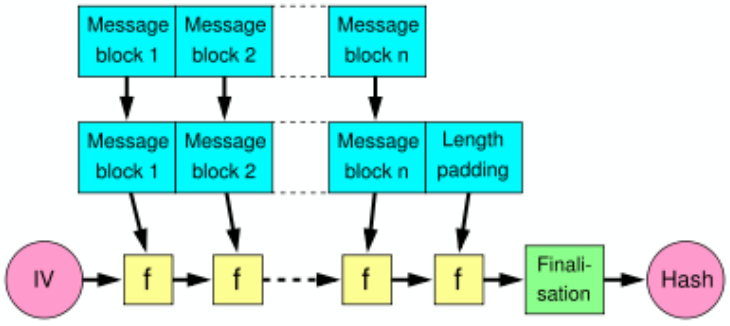


Figure 23: *[5] The Merkle-Damgård hash construction.*

The main building blocks of the Merkle-Damgård structure are:

IV: Initialization Vector or Initial Value is a fixed value used as the chaining variable for the very first iteration.

f: the compression function or one-way hash function which is either specially designed for hashing or based on a block cipher. The compression function generally takes an input of fixed length and produces an output of fixed length.

Finalisation: an output transformation function which usually reduces further the length of the output value of the last iteration.

Hash: the message digest or the hash result.

As we can see from the figure 23, the entire message to be hashed is first divided into n blocks of equal length. The actual length of the message blocks depends on the requirements set by the compression function f. The message is then padded, always, such that its length is a multiple of some specific number. The padding is done by adding after the last bit of the last message block a single 1-bit followed by the necessary number of 0-bits. The length padding which consists of appending a k-bit representation the length in bits of the original message (that is, the message before any padding has been applied) takes place in such a way that the padding length bits are added as the last bits of the padded message block prior to being processed by the compression function. Every block is processed by the compression function in the same iterative manner.

The compression function always takes two inputs in each step or iteration, a message block and a chaining variable.
In the first iteration, the chaining variable is the IV or Initialization Vector. It is given, together with the first block of message, as inputs to the compression function. The output of the compression function f in the first iteration is the chaining variable in the second iteration. The output of the compression function f in the ith iteration is the chaining variable in the $(i+1)th$ iteration and so on until we reach the last iteration.
In the last iteration, the output of the compression function is used as an input to a finalization function which reduces further the length of the final output value from the compression function (however, in some cases the finalization function is not present and the output value of the compression function f in the last iteration is used as the final hash result).
In general, for a message M consisting of t blocks $M_0, M_1, ..., M_{t-1}$, the computation of the message digest can be defined as follows:

$H_0 = IV$,
$H_{i+1} = f(H_i, M_i) \quad$ for $0 \leq i < t$
$H(M) = Finalisation(H_t)$

In a pseudo code, the same computation can be defined as follows:

$computeDigest(M)$ {
$M_{0\cdots t-1} = divBlock(M)$
$H_0 = IV$
$for(i = 0; i < t; i++)$ {
$H_{i+1} = f(H_i, M_i)$
}
$H(M) = Finalisation(H_t)$
$return\ H(M)$
}

Here the function *computeDigest*() takes the message M as input and returns as output the hash result H (M). The inner function *divBlock*() breaks up the message M into t blocks of equal length and returns an array consisting of the t message blocks. The details of these functions are deliberately overlooked for the sake of simplicity. We now examine some of the standards hash functions in more details.

4.2 The MD5 hash Algorithm

The MD5 (1992) message-digest algorithm was designed as a strengthened extension of the MD4 (1990) message digest algorithm. MD5 is slightly slower than MD4; this is a classical example where security is favoured at the expense of speed. Both algorithms were developed by Ron Rivest who is the "R" in the RSA [Rivest-Shamir-Adleman] public-key encryption algorithm.

4.2.1 Description of the MD5 algorithm

The algorithm accepts an input message of arbitrary length and produces a 128-bit "message digest", "fingerprint" or "hash result". Figure 24 depicts the way the input message is turned into a 128-bit message digest.

The actual processing of the MD5 algorithm consists of the following 5 steps:

Step 1: Append padding bits

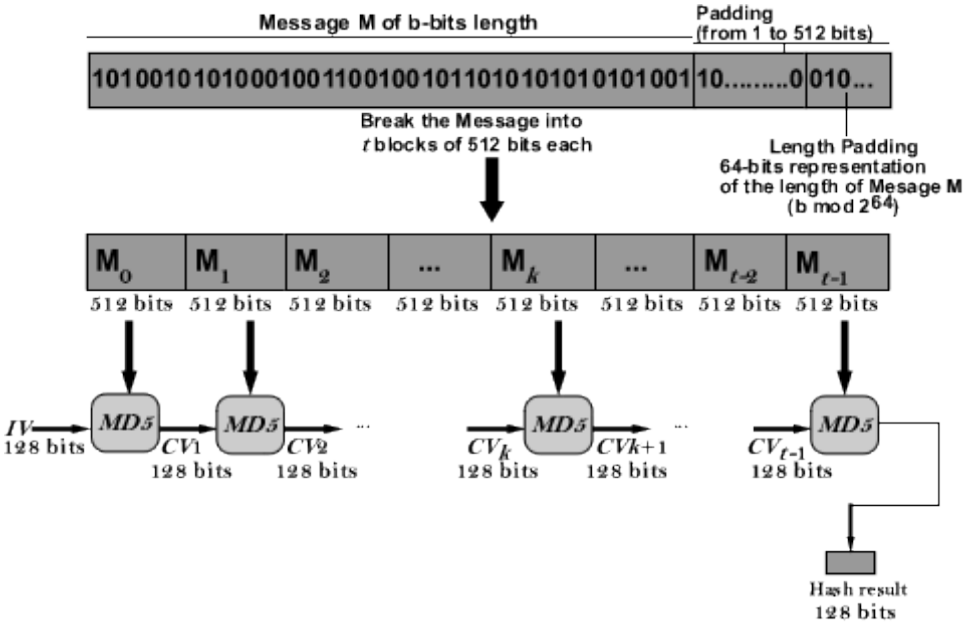


Figure 24: *The MD5 algorithm.*

During this step, the message is extended or padded in such a way that its total length in bits is congruent to 448 *modulo* 512. This operation is always performed even if the message's length in bit is originally congruent to 448 *modulo* 512. We notice that 448 + 64 = 512, so the message is padded such that its length is now 64 *bits* less an integer multiple of 512.

Padding is done by appending to the message a single "1" bit followed by the necessary amount of "0" bits so that the length in bits of the padded message becomes congruent to 448 *modulo* 512. For example, if the message is 447 bits long, it is padded by 1 bit to a length of 448 *bits* (the single bit 1 is appended to the end of the message in this particular case). On the other hand, if the message is 448 *bits* long, it is padded by 512 *bits* to a length of 960 *bits*. Thus, at least 1 bit and at most 512 bits are appended during this step.

Step 2: Append length

A $64 - bit$ representation of the length in bits of the original message M (before the padding bits were added) is appended to the result of step 1. Here, there is a little trick in representing the length of message M in the case where it is greater than 2^{64}. If the length of the original message is indeed greater than 2^{64} (= 184 467 440 73 709 551 616), then only the low order $64 - bits$ of the length of message M are used. Hence, the field contains the length of the original message M modulo

2^{64}. These bits are appended as two $32-bit$ words and appended low-order (least significant) word first.
The result of step 1 and step 2 is a message with a length in bits that is an integer multiple of 512 bits. Consequently, the result has a length that is also a multiple of 16 $32-bits$ word. In the figure above, the padded message is represented as a sequence of t $512-bit$ blocks $M_0, M_1, \cdots, M_{t-1}$. Therefore, the total length in bits of the expanded message is $t \times 512$. Let $M[0 \cdots N-1]$ denotes the words of the resulting message where t is a multiple of 16. This means that $N = t \times 16$. This is easily appreciated when we know that in each block, there are 16 $32-bit$ words and that we have t blocks in total.

Assuming that we are given a message M with a length in bit of 447. Step 1 tells us that a single bit will be added to that message to bring its length to 448. Step 2 appends a 64-bit representation of the length of the original message (447) which is less than 2^{64} and we append the following $64-bit$ representation of the decimal number 447: 00000000000000000000000000000000000000000000000000000000110111111

Step 3: Initialize MD buffer

A $128-bit$ ($4 \times 32-bit$) buffer (A, B, C, D) is used to hold intermediate and final result of the MD5 hash algorithm. These registers are initialized to the following $32-bit$ values in hexadecimal:

$$\begin{array}{lcl} \text{A} & = & 67\ 45\ 23\ 01 \\ \text{B} & = & \text{ef cd ab 89} \\ \text{C} & = & 98\ \text{ba dc fe} \\ \text{D} & = & 10\ 32\ 54\ 76 \end{array}$$

These values are stored in little-endian format, meaning that the low -order bytes of a word is placed in the low-address byte position. The initialization values appear then as follows:

$$\begin{array}{lcl} \text{word A} & = & 01\ 23\ 45\ 67 \\ \text{word B} & = & 89\ \text{ab cd ef} \\ \text{word C} & = & \text{fe dc ba } 98 \\ \text{word D} & = & 76\ 54\ 32\ 10 \end{array}$$

These four variables (they are indeed variables since they change value) are copied into different variables: A is saved as AA, B as BB, C as CC and D is saved as DD.

Step 4: Define four auxiliary functions and process message

This step consists of sixty-four (64) steps divided into four (4) rounds of processing. The four rounds are almost identical, with the main difference being that each round uses a different primitive logical function, denoted by F, G, H, and I in the specification. Let us first define the four functions. We note that each of them takes three 32-bit words as input and yields one 32-bit word as output.

Round	*Primitive function*	*Steps*(64)
1	$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$	$0 \leq j \leq 15$
2	$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z)$	$16 \leq j \leq 31$
3	$H(X,Y,Z) = X \oplus Y \oplus Z$	$32 \leq j \leq 47$
4	$I(X,Y,Z) = Y \oplus (X \vee \neg Z)$	$48 \leq j \leq 63$

Table 1: *The primitive functions of the MD5 compression algorithm.*

where

$\wedge = AND$, $\vee = OR$, $\oplus = XOR$, $\neg X = NOT(X)$

Each round takes as input the current 512-bit message block M_k and the 128-bit buffer value ABCD and produces as output an updated value of the buffer earlier referred to as the chaining variable CV_k. In addition, each round also uses one-fourth of a 64-element table denoted $T[1 \cdots 64]$ which is constructed from the sine function. It is constructed such that the $i-th$ element of the table T, denoted $T[i]$, is equal to the integer part of $232 \times abs(sin(i))$, where i is in radian. Since $abs \times (sin(i))$ is a number between 0 and 1, the elements of the table T are numbers less than or equal to 2^{32}, Hence, they can be represented in 32 bits. The values in 32-bits representation are listed in the table below:

Step 5: Output

The output from the very last round is the 128-bit hash result or message digest we obtain after we have incrementally processed all t 512-bit blocks of the message. The entire process can be summarized as follows:

$$\begin{aligned} CV_0 &= IV \\ CV_{k+1} &= SUM_{32}(CV_k, RF_I[M_k, RF_H[M_k, RF_G[M_k, RF_F[M_k, CV_k]]]]) \\ MD5_{SUM} &= CV_t \end{aligned}$$

where

$$\begin{aligned} IV &= \text{the initial value of the ABCD buffer, defined by step 3} \\ M_k &= \text{the } k-th \text{ 512-bit block of the message} \\ CV_k &= \text{the chaining variable processed with the } k-th \text{ block of message} \\ RF_x &= \text{the round function using primitive logical function } x \\ MD5_{SUM} &= \text{the final hash result or message digest} \\ SUM_{32} &= \text{addition } modulo\ 2^{32} \end{aligned}$$

$T[1] = d76aa478$	$T[17] = f61e2562$	$T[33] = fffa3942$	$T[49] = f4292244$
$T[2] = e8c7b756$	$T[18] = c040b340$	$T[34] = 8771f681$	$T[50] = 432aff97$
$T[3] = 242070db$	$T[19] = 265e5a51$	$T[35] = 6d9d6122$	$T[51] = ab9423a7$
$T[4] = c1bdceee$	$T[20] = e9b6c7aa$	$T[36] = fde5380c$	$T[52] = fc93a039$
$T[5] = f57c0faf$	$T[21] = d62f105d$	$T[37] = a4beea44$	$T[53] = 655b59c3$
$T[6] = 4787c62a$	$T[22] = 02441453$	$T[38] = 4bdecfa9$	$T[54] = 8f0ccc92$
$T[7] = a8304613$	$T[23] = d8a1e681$	$T[39] = f6bb4b60$	$T[55] = ffeff47d$
$T[8] = fd469501$	$T[24] = e7d3fbc8$	$T[40] = bebfbc70$	$T[56] = 85845dd1$
$T[9] = 698098d8$	$T[25] = 21e1cde6$	$T[41] = 289b7ec6$	$T[57] = 6fa87e4f$
$T[10] = 8b44f7af$	$T[26] = c33707d6$	$T[42] = eaa127fa$	$T[58] = fe2ce6e0$
$T[11] = ffff5bb1$	$T[27] = f4d50d87$	$T[43] = d4ef3085$	$T[59] = a3014314$
$T[12] = 895cd7be$	$T[28] = 455a14ed$	$T[44] = 4881d05$	$T[60] = 4e0811a1$
$T[13] = 6b901122$	$T[29] = a9e3e905$	$T[45] = d9d4d039$	$T[61] = f7537e82$
$T[14] = fd987193$	$T[30] = fcefa3f8$	$T[46] = e6db99e5$	$T[62] = bd3af235$
$T[15] = a679438e$	$T[31] = 676f02d9$	$T[47] = 1fa27cf8$	$T[63] = 2ad7d2bb$
$T[16] = 49b40821$	$T[32] = 8d2a4c8a$	$T[48] = c4ac5665$	$T[64] = eb86d391$
Round 1	Round 2	Round 3	Round 4

Table 2: *The 32-bit word matrix T and the round in which its values are used*

The final message digest is stored in the ABCD buffer (see figure 25). It is output by beginning with the low order byte of A and ending with the high order byte of D.

The output of round 4 is added to the chaining variable of the previous round to produce the next chaining variable. The addition is *modulo* 2^{32} and it is performed separately on each of the four words in the buffer.
We now look at the inner structure of the MD5 compression function labelled MD5 in figure captioned The MD5 algorithm.

4.2.2 The MD5 Compression Function

We have just mentioned that the entire MD5 algorithm consists of four rounds and each round, in turn, consists of 16 steps which give us 64 steps in total. Each step takes the following general form:

$A \leftarrow B + [(A + PF(B, C, D) + M[k] + T[i]) <<< s]$

with

A, B, C, D	=	the four words of the buffer
PF	=	one of the primitive functions F, G, H, I
$M[k]$	=	the kth 32-bit word in the 512-bit message block
$T[i]$	=	the ith 32-bit word in table T
$+$	=	addition *modulo* 2^{32}
$<<< s$	=	the circular left shift of the 32-bit argument by s bits

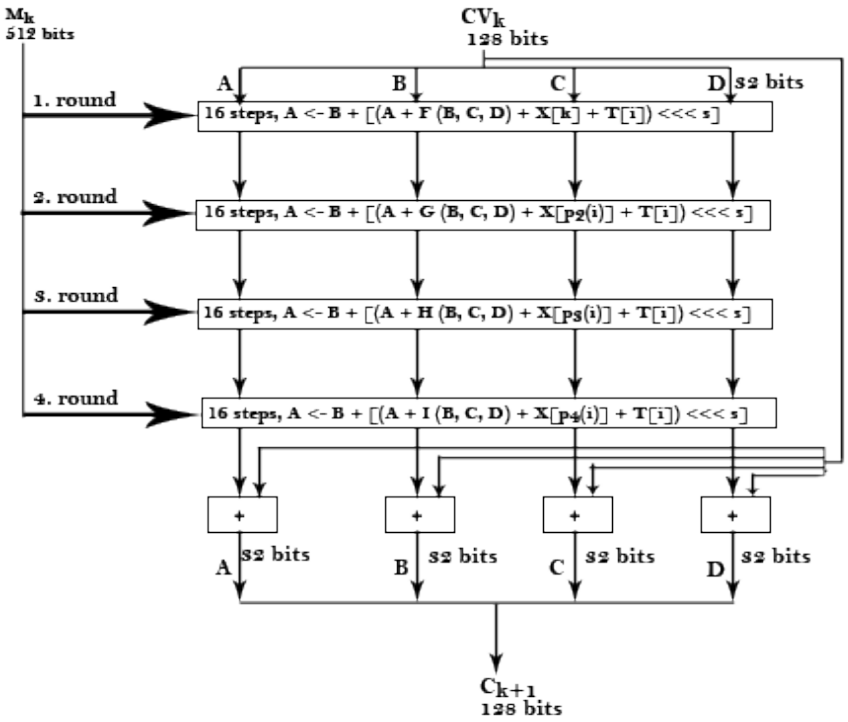


Figure 25: *The MD5 compression of a single 512-bit message block.*

The four words A, B, C, D of the buffer are used in such a way that produces a word-level circular right shift of one word after every step. After each step, we update one of the 4 bytes of the ABCD buffer. Knowing that we have 16 steps, it results that each 32-bit word of the buffer is updated four times at the end of the fourth round and an additional fifth time to produce the final output (chaining variable) for the current block.

In round 1, the primitive function F is used. The 16 32-bits words of the 512-bit message block are use in their original order $M[0]$ through $M[15]$. Each of the 16 words is used only once in each step.

In round 2, the primitive function G is used. The 16 32-bits words of the 512-bit message

block are used in the following order:

$p_2(i) = (1 + 5i)\ mod\ 16$

In round 3, the primitive function H is used. The 16 32-bits words of the 512-bit message block are used in the following order:

$p_3(i) = (5 + 3i)\ mod\ 16$

In round 4, the primitive function I is used. The 16 32-bits words of the 512-bit message block are used in the following order:

$p_4(i) = 7i\ mod\ 16$

We note that every element of the table T is used exactly once during one MD5 round which consists of four rounds where each of the four primitive function is used.

Figure 26 shows a single step of the MD5 operation. As we can see, one of the buffers gets updated after every step. This basically means that each of the four buffer A, B, C, D changes its content as each of them moves to the left side of the general form $A \leftarrow B + [(A + PF(B, C, D) + M[k] + T[i]) <<< s]$ in turn after every single step.

4.2.3 Security of MD5

The MD5 algorithm has one interesting property that every bit of the output is a function of every bit of the input. The complexity in the repeated use of the primitive functions and the additive constant $T[i]$ together with the circular left shifts unique to every round produce a well mixed hash result.
This procedure makes it very unlikely that two messages that show a similar regularity will have the same hash result.

Ron Rivest conjectures that MD5 is as strong as possible for a 128-bit hash code. What Rivest means is that finding two messages having the same hash value requires 2^{64} operations. This is due to the birthday paradox explained earlier; and finding a message given its corresponding message digest will take 2^{128} operations, this is known as the pre-image attack.
However, this is only an opinion which is not proven and any mechanism which takes less operations than that will prove the MD5 algorithm less secure than originally believed.

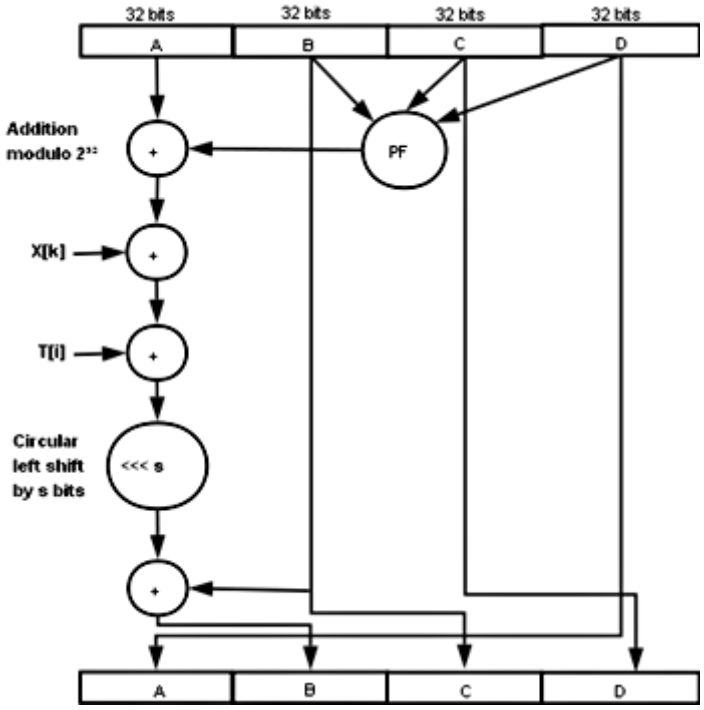


Figure 26: *A single MD5 step operation.*

4.2.4 Attacks on MD5

More generally, we can distinguish three main categories of attacks on cryptographic hash function which are **pre-image** attacks, **second pre-image** attacks and **collision** attacks. In one way or another, all kind of attacks fall into either one of these categories. This is the main reason why one-way hash functions are required to be pre-image resistant and second pre-image resistant while in addition to these two properties, collision-resistant hash function need to exhibit the property of being collision-resistant. We take a look at some commonly known attacks.

Collision attack in MD5 cryptographic hash function

In August 2004 Xiaoyun Wang and Hongbo Yu of Shandong University in China published an article[15] in which they describe an algorithm that can find two different sequences of 128 bytes with the same MD5 hash. Their research was motivated by the possibility of finding a colliding pair of messages, each consisting of two blocks. The attack revolves around finding two distinct message blocks (M_0, N_0) and (M_1, N_1) where the first blocks differ only in a predefined constant vector cv_1 such that $M_1 = M_0 + cv_1$ and the second message blocks differs in a predefined constant vector cv_2

$(cv_2 = -cv_1 \ modulo \ 2^{32})$ such that $N_1 = N_0 + cv_2$ and $MD5(M_0, N_0) = MD5(M_1, N_1)$.

However, there is a considerable amount of conditions that have to be met for the attack to be successful. Basically, the attack makes use of differential or differences in message that are spread over a length of two message blocks. The first block's difference introduces a small difference in the original state or initialization vector whereas the second block's difference cancels out the difference introduced by the first block. We also note that finding the first blocks (M_0, M_1) takes about 2^{39} MD5 operations, and finding the second blocks (N_0, N_1) takes about 2^{32} MD5 operations. The application of this attack on IBM P690 takes about one hour to find M_0 and M_1, where in the fastest cases it takes only 15 minutes. Then, it takes only between 15 seconds to 5 minutes to find the second blocks N_0 and N_1.

Pure Brute-force Attack

The pure brute-force attack is one in which all possible words of a certain length are tried until the correct one is found. This attack is guarantied to work, that is why one ususally chooses the length of the hash result in such a way that the brute-force attack becomes impractical or too slow and thus less attractive.

4.3 The Secure Hash Algorithm - SHA

The Secure Hash Algorithm (SHA) was developed by the National Security Agency (NSA) and published in 1993 by the National Institute of Standard and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS PUB 180). SHA is based on and shares the same building blocks as the MD4 algorithm. Unlike the MD4 algorithm, the design of SHA introduced a new procedure which expands the 16-word message block input to the compression function to an 80-word block among other things. In 1994, NIST announced that a technical flaw in SHA was found. And that this flaw makes the algorithm less secure than originally believed. No further details were given to the public, only that a small modification was made to the algorithm which was now known as SHA-1 and published in FIBS PUB 180-1.

The SHA-2 family of hash algorithm consists of five cryptographic hash functions denoted by SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512. The last four variants are sometimes collectively referred to as SHA-2. In fact, NIST updated its hash function standard to FIPS PUB 180-2 in 2002. This update specified three new hash functions, next to SHA-1, known as SHA-256, SHA-384, and SHA-512. SHA-1 is designed to produce a 160-bit message digest. The other hash functions have the length of their message digest indicated by the number following prefix "SHA-"; thus SHA-256 produces a 256-bit message digest, whereas SHA-384 produces a 384-bit hash value and so on.

SHA-224 which produces a message digest of 224-bit was added to the standard in 2004. We describe in more details the SHA-1 algorithm.

4.3.1 Description of the SHA-1 algorithm

The SHA-1 algorithm accepts as input a message with a maximum length of $2^{64} - 1$ and produces a 160-bit message digest as output. The message is processed by the compression function in 512-bit block. Each block is divided further into sixteen 32-bit words denoted by M_t for $t = 0, 1, \cdots, 15$. The compression function consists of four rounds, each round is made up of a sequence of twenty steps. A complete SHA-1 round consists of eighty steps where a block length of 512 bits is used together with a 160-bit chaining variable to finally produce a 160-bit hash value. The processing works as described in the following steps:

Step 1: Append padding bits

The original message is padded so that its length is congruent to $448 \, modulo \, 512$. Again, padding is always added although the message already has the desired length. Padding consists of a single 1 followed by the necessary number of 0 bits.

Step 2: Append length

A 64-bit block treated as an unsigned 64-bit integer (most significant byte first), and representing the length of the original message (before padding in step 1), is appended to the message. The entire message's length is now a multiple of 512.

Step 3: Initialize the buffer

The buffer consists of five (5) registers of 32 bits each denoted by A, B, C, D, and E. This 160-bit buffer is used to hold temporary and final results of the compression function. These five registers are initialized to the following 32-bit integers (in hexadecimal notation).

$$\begin{array}{lcl} \text{A} & = & 67\ 45\ 23\ 01 \\ \text{B} & = & ef\ cd\ ab\ 89 \\ \text{C} & = & 98\ ba\ dc\ fe \\ \text{D} & = & 10\ 32\ 54\ 76 \\ \text{E} & = & c3\ d2\ e1\ f0 \end{array}$$

We can see that the registers A, B, C, and D are exactly the same as the four registers

used in MD5 algorithm. But in SHA-1, these values are stored in big-endian format, which means that the most significant byte of the word is placed in the low-address byte position. Hence the initialization values (in hexadecimal notation) appear as follows:

$$\begin{array}{lcl} \text{word A} & = & \text{67 45 23 01} \\ \text{word B} & = & \text{ef cd ab 89} \\ \text{word C} & = & \text{98 ba dc fe} \\ \text{word D} & = & \text{10 32 54 76} \\ \text{word E} & = & \text{c3 d2 e1 f0} \end{array}$$

Step 4: Process message in 512-bit blocks

The compression function is divided into twenty sequential steps composed of four rounds of processing where each round is made up of twenty steps. The four rounds are structurally similar to one another with the only difference that each round uses a different Boolean function, which we refer to as f_1, f_2, f_3, f_4 and one of four different additive constants K_t $(0 \leq t \leq 79)$ which depends on the step under consideration. The values of the four dictinct additives constant are given in table 3 below.

Step number	Hexadecimal notation
$0 \leq t \leq 19$	$K_t = 5a827999$
$20 \leq t \leq 39$	$K_t = 6ed9eba1$
$40 \leq t \leq 59$	$K_t = 8f1bbcdc$
$60 \leq t \leq 79$	$K_t = ca62c1d6$

Table 3: *The four additive constants used in SHA-1 algorithm*

Every step updates two of the five registers. The step operation which updates the value of the E register and rotates the value of the B register by 30 bit position to the left is of the following form:
$A, B, C, D, E \leftarrow (E + f_r(t, B, C, D) + [A <<< 5] + M_t + K_t), A, [B <<< 30], C, D$

where

A, B, C, D, E	=	the five registers of the SHA-1 buffer
t	=	the step number, $0 \leq t \leq 79$
f_r	=	the primitive logical function used in step t and round r
$<<< s$	=	the circular left shift of the 32-bit word by s bits
M_t	=	a 32-bit word derived from the current 512-bit input block
K_t	=	one of four additive constants
$+$	=	addition *modulo* 2^{32}

Each Boolean function takes three 32-bit words as input and produces a 32-bit word as output. Each function performs a different set of bitwise logical operation on its inputs such that the *nth* bit of the output is a function of the *nth* bit of all three inputs.

We have said earlier that SHA-0 is similar to SHA-1 with the only difference residing in the procedure which expands the 16-word message block input to an 80-word message block. This procedure is defined in SHA-0 as:

$$M_t = M_{t-16} \oplus M_{t-14} \oplus M_{t-8} \oplus M_{t-3}$$

whereas it is defined in SHA-1 as:

$$M_t = (M_{t-16} \oplus M_{t-14} \oplus M_{t-8} \oplus M_{t-3}) <<< 1$$

As it can be seen, SHA-1 adds a left circular shift by 1 bit position "<<< 1".

Step 5: Output

After processing the last 512-bit message block t (assuming that the message is divided into t 512-bit blocks), we obtain a 160-bit message digest. The compression function uses a feedforward operation where the chaining variable CV_k input to the first round is added to the output obtained after execution of step 80 to produce the next chaining variable CV_{k+1}. This addition is performed *modulo* 2^{32} and independently for each of the five words in the buffer. We describe this operation as follows:

$$CV_0 = IV$$
$$CV_{k+1} = S_{32}\left(CV_k, f_4\left[M_{60\cdots79}, f_3\left[M_{40\cdots59}, f_2\left[M_{20\cdots39}, f_1\left[M_{0\cdots19}, CV_k, K_{0\cdots19}\right], K_{20\cdots39}\right], K_{40\cdots59}\right], K_{60\cdots79}\right]\right)$$
$$H_r = CV_t$$

where

IV	=	initial value of the ABCDE buffer, used to deal with the first block in a chaining mode
$f1[\cdots]$	=	output of the first round consisting of 20 steps
$f2[\cdots]$	=	output of the second round
$f3[\cdots]$	=	output of the third round
$f4[\cdots]$	=	output of the fourth round
S_{32}	=	addition *modulo* 2^{32}
H_r	=	the final hash result or message digest

The primitive logical functions f_r are defined as shown in table 4.

with the following correspondence between the logical operators as shown in table 5

Step	Function	Value
$0 \leq t \leq 19$	$f_1 = f(t, B, C, D)$	$(B \wedge C) \vee (\neg B \wedge D)$
$20 \leq t \leq 39$	$f_2 = f(t, B, C, D)$	$B \oplus C \oplus D$
$40 \leq t \leq 59$	$f_3 = f(t, B, C, D)$	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
$60 \leq t \leq 79$	$f_4 = f(t, B, C, D)$	$B \oplus C \oplus D$

Table 4: *The four primitive logical functions used in SHA-1 algorithm*

Operand	Name	Description
$\wedge$	AND	Is true if both values are true
$\vee$	OR	Is true if either value is true
$\oplus$	XOR	Is true if either value is true, not both
$\neg$	NOT	Is true if the value is not true

Table 5: *The description of the logical operators*

As compared to MD5, only the function used in the first round has not changed. Otherwise, the remaining three functions for the last three rounds of SHA-1 are not the same as the one used in the MD5 compression algorithm. We notice that in the SHA-1 algorithm, the functions f2 and f4 have the same structure and are used respectively in round 2 and 4. These two functions are structurally equivalent to the function H used in the round 3 of the MD5 compression function. However, the function f3 of the SHA-1 algorithm is entirely new. We now take a closer look at the way SHA-1 expands the 16 block words to 80 words used by the compression function.

Deriving 80 32-bit word values from one 512-bit message block

Each 512-bit message block comprises 16 32-bit words ($16 \times 32 = 512$). During the step which processes the message in 512-bit blocks, the first 16 words of every message block is taken and used directly as it appears. The additional 64 blocks are derived by following the algorithm given by:

$$M_t = (M_{t-16} \oplus M_{t-14} \oplus M_{t-8} \oplus M_{t-3}) <<< 1$$

This means that if we assume that word M_0 through M_{15} represent the first 16 words (used in the first 16 steps), then for the step 17 the word M_{16} is given by:

$$M_{16} = (M_0 \oplus M_2 \oplus M_8 \oplus M_{13}) <<< 1$$

Up to step 80 where M_{79} is given by:

$$M_{79} = (M_{63} \oplus M_{65} \oplus M_{71} \oplus M_{76}) <<< 1$$

As we can see, the value of all subsequent 16 words block M_t are obtained by XOR-ing four of the preceding values of M_t and by applying a circular left shift by one bit to the resulting value.

Compared to MD5, this procedure is entirely new. Recall that in the MD5 algorithm, an array of 32-bit words $M[0 \cdots 15]$ holds the value of the current 512-bit input block being processed. In each round, which consists of 16 steps, each of the 16 words of $M[0 \cdots 15]$ is used exactly once. Hence, each step uses one of the 16 32-bit words of the current 512-bit input block. The order in which these words is used varies from round to round except that in the first round the words are used in their original order, that is the 32-bit words $M[0], M[1], \cdots, M[15]$ are used respectively in step $1, 2, \cdots, 16$. For round 2 through round 4, MD5 introduced a special permutation which assigns each of the sixteen 32-bit word $M[0 \cdots 15]$ to a particular step within the remaining rounds.

The word expansion introduced by SHA-1 augments the interdependency between every message block and the final message digest. Together with the longer output of 160-bit message digest, SHA-1 simply strengthens the one-wayness, pre-image resistance, second pre-image resistance and collision resistance of the SHA-1 viewed as cryptographic hash function.

4.3.2 Security of SHA-1

Pre-image resistance

The difficulty of producing any message having a given message digest is of the order of 2^{160}.

Second pre-image resistance

The difficulty of producing two distinct messages having the same message digest is of the order of 2^{80} operations.

Speed, simplicity and architecture

Addition in SHA-1 is perfomed *modulo* 2^{32}, thus it is well suited for a 32-bit architecture. Because SHA-1 involves 80 steps and must process 160-bit buffer (five 32-bit registers), it is slower than MD5 on the same architecture. SHA-1 is

also simple to describe and to implement in both software and hardware. Finally SHA-1 uses a big-endian scheme for interpreting a message as a sequence of 32-bit words.

4.3.3 Attacks against SHA-1

We know that SHA-1 produces a 160-bit message digest. If one cannot find collision in less than 2^{80} operations, then SHA-1 is considered secure and can still be used. But recently, a group of chinese cryptographers[21] were able to find collisions in SHA-1 in 2^{69} calculations. This is about 2000 times faster than a brute-force search attack.

4.4 The RIPEMD-160 Algorithm

The first RIPEMD hash function was degined in 1992 under the European RIPE (RACE Integrity Primitives Evaluation) project. It is a function that produced a 128-bit hash value and had its design based on the MD4 algorithm. In constrast to MD4, RIPEMD's compression function consists of two slightly modified versions of the MD4 compression function which are executed in parallel.

Due to a successful attack on two rounds of the RIPEMD by an outsider (a person not member of the RIPE project group), namely H. Dobbertin, some members of the RIPE consortium decided not only to upgrade RIPEMD, but also to include H. Dobbertin in the team!

4.4.1 Description of RIPEMD-160 Algorithm

The algorithm takes as input a message of arbitrary length and computes a hash result of 160 bits. The input message is processed 512-bit block at a time. The compression function consists of two parallel trails of a modified version of the MD5 algorithm. Each trail consists of five rounds, whereas each round is composed of sixteen sequential steps. A complete RIPEMD-160 round consists of eighty parallel steps where a message block of 512 bits is used together with a 160-bit chaining variable to finally produce a 160-bit hash value. The processing works as described in the following steps:

Step 1: Append padding bits

The message is padded such that its length is congruent to 448 *modulo* 512. The process of padding consists of appending a single 1-bit followed by the necessary

number of 0-bit right after the last bit of the original message. Padding is always added, even if the message's length is already congruent to 448 *modulo* 512. Thus, the number of padding bits ranges from 1 to 512.

Step 2: Append length

A block of 64 bits representing the length of the original message (before padding step) is appended to the message after the last bit of the padded message. This block is treated as an unsigned 64-bit integer (least signicant byte comes first). RIPEMD-160 adopts the little-endian convention for interpreting a message as a sequence of 32 bit words.

Step 3: Initialize the RIPEMD-160 buffer

A 160-bit buffer is used to hold intermediate and final results of the RIPEMD-160 hash function. The buffer consists of five 32-bit registers (A, B, C, D, and E). The registers are initialized to the following hexadecimal values.

$$\begin{array}{lcl} \text{A} & = & \text{67 45 23 01} \\ \text{B} & = & \text{ef cd ab 89} \\ \text{C} & = & \text{98 ba dc fe} \\ \text{D} & = & \text{10 32 54 76} \\ \text{E} & = & \text{c3 d2 e1 f0} \end{array}$$

The values are the same as those used in SHA-1, but in contrast to SHA-1, they are stored in little-endian format which means the low-order byte (least significant byte) of a word is stored in the low-address byte position. Hence, the registers appear in memory as follows:

$$\begin{array}{lcl} \text{word A} & = & \text{67 45 23 01} \\ \text{word B} & = & \text{ef cd ab 89} \\ \text{word C} & = & \text{98 ba dc fe} \\ \text{word D} & = & \text{10 32 54 76} \\ \text{word E} & = & \text{c3 d2 e1 f0} \end{array}$$

Step 4: Process message in 512-bit blocks

The RIPEMD-160 algorithm comprises ten rounds of sixteen steps each. The ten rounds are arranged as two separate trails (left and right) of five rounds each. The

ten rounds have a similar but not an identical structure. Each round uses one of five primitive logical functions (Boolean function) denoted by f_1, f_2, f_3, f_4, f_5. The functions are used from f_1 through f_5 in the left trail and in reverse order from f_5 through f_1 in the right trail. We denote the registers of the left trail by A_L, B_L, C_L, D_L, E_L and the registers of the right trail by A_R, B_R, C_R, D_R, E_R. The initial chaining variable is denoted by A_0, B_0, C_0, D_0, E_0.
Each round takes as input the current 512-bit block and the 160-bit buffer value A_L, B_L, C_L, D_L, E_L (left trail) or A_R, B_R, C_R, D_R, E_R (right trail) and updates the value of the register in the left or right trail respectively. An additive constant K_r which depends on the round and on the trail is also used. There are nine distinct additive constants in total, including zero which is used in round 1 on the left trail and in round 5 on the right trail. Table 6 summarizes the values of the additive constants in hexadecimal.

	Left trail	Right trail
Step number	Hexadecimal value	Hexadecimal value
$0 \leq t \leq 15$	$K_1 = 00000000$	$K_1 = 50a28be6$
$16 \leq t \leq 31$	$K_2 = 5a827999$	$K_2 = 5c4dd124$
$32 \leq t \leq 47$	$K_3 = 6ed9eba1$	$K_2 = 6d703ef3$
$48 \leq t \leq 63$	$K_4 = 8f1bbcdc$	$K_2 = 7a6d76e9$
$64 \leq t \leq 79$	$K_5 = a953fd4e$	$K_2 = 00000000$

Table 6: *The four additive constants used in RIPEMD-160 algorithm*

Every step updates the value of two of the five registers. Five consecutive steps update the value of the registers A, E, D, C, B in this order and also rotate respectively the values of the registers C, B, A, E, and D by ten bit positions to the left. Thus, in both trails, the five registers are updated sixteen times after the last step of the last round.
The output value of the fifth round is added to the input of the first round (CV_q) to produce CV_{q+1}. The addition is perfomed $modulo\, 2^{32}$ and involves three words coming from the chaining variable CV_q, the left trail and the right trail as follows:

$$CV_{q+1}(0) = CV_q(1) + C_L + D_R$$
$$CV_{q+1}(1) = CV_q(2) + D_L + E_R$$
$$CV_{q+1}(2) = CV_q(3) + E_L + A_R$$
$$CV_{q+1}(3) = CV_q(4) + A_L + B_R$$
$$CV_{q+1}(4) = CV_q(0) + B_L + C_R$$

where

A_L	=	value of the register A in the left trail. It also applies to B_L, C_L, D_L, and E_L that is B_L denotes the value of register B in the left trail.
A_R	=	value of the register A in the right trail. It also applies to B_R, C_R, D_R, and E_R that is B_R denotes the value of register B in the right trail.
$CV_q(0)$	=	A_0, the value of the register A in the chaining variable input to round 1
$CV_q(1)$	=	B_0, the value of the register B in the chaining variable input to round 1
$CV_q(2)$	=	C_0, the value of the register C in the chaining variable input to round 1
$CV_q(3)$	=	D_0, the value of the register D in the chaining variable input to round 1
$CV_q(4)$	=	E_0, the value of the register E in the chaining variable input to round 1
$CV_{q+1}(0)$	=	the value of the register A in the next chaining variable
$CV_{q+1}(1)$	=	the value of the register B in the next chaining variable
$CV_{q+1}(2)$	=	the value of the register C in the next chaining variable
$CV_{q+1}(3)$	=	the value of the register D in the next chaining variable
$CV_{q+1}(4)$	=	the value of the register E in the next chaining variable

Step 5: Output the message digest

The message digest is the final output after the last 512-bit message block has been processed.

4.4.2 The RIPEMD-160 compression function

We have mentioned above that the RIPEMD-160 algorithm consists of two parallel lines or trails. Each trail is made up of five rounds which in turn consist of a sequence of 16 steps each. Thus, the compression function in each trail comprises eighty sequential steps. Each round, in both the left and the right trail, uses one of five primitive logical functions in such a way that the order in which they are used in the left trail is reversed in the right trail as shown in the table 7 :

Trail	Round 1	Round 2	Round 3	Round 4	Round 5
Left	f_1	f_2	f_3	f_4	f_5
Right	f_5	f_4	f_3	f_2	f_1

Table 7: *The order and round in which the primitive functions are used in both trails*

Like in the MD5 and the SHA-1 algorithm, each primitive function takes as input three 32-bit words and produces one 32-bit word as output. The five functions are summarized in the table 8:

As before, every 512-bit message block is held in an array of sixteen 32-bit words represented by $M[0 \cdots 15]$. Each of these 16 words is used simultaneously in both

Step	Function	Value
$0 \leq t \leq 15$	$f_1 = f(t, B, C, D)$	$B \oplus C \oplus D$
$16 \leq t \leq 31$	$f_2 = f(t, B, C, D)$	$(B \wedge C) \vee (\neg B \wedge D)$
$32 \leq t \leq 47$	$f_3 = f(t, B, C, D)$	$(B \wedge \neg C) \oplus D$
$48 \leq t \leq 63$	$f_4 = f(t, B, C, D)$	$(B \wedge D) \vee (C \wedge \neg D)$
$64 \leq t \leq 79$	$f_5 = f(t, B, C, D)$	$B \oplus (C \vee \neg D)$

Table 8: *The four primitive logical functions used in RIPEMD-160 algorithm*

the left and the right trail during every step. However, the 16 words are not used in the same order, neither in each round nor in each trail. The order in which the words are used relies on two permutations and depends on the round and on the trail under consideration.
We define the permutation ρ as follows:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(i)$	7	4	13	1	10	6	15	3	12	0	9	5	2	14	11	8

We define the second permutation π which is given by $\pi(i) = 9i + 5(modulo\ 16)$ as follows:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(i)$	5	14	7	0	9	2	11	4	13	6	15	8	1	10	3	12

And the order of the message, depending on the round and on the trail, is summarized in table 9:

Trail	Round 1	Round 2	Round 3	Round 4	Round 5
Left	Identity	ρ	ρ^2	ρ^3	ρ^4
Right	π	$\rho\pi$	$\rho^2\pi$	$\rho^3\pi$	$\rho^4\pi$

Table 9: *Permutation of message words in RIPEMD-160 algorithm.*

This simply means that for the first round and in the left trail, the 16 32-bit words are use from 0 through 15 in that order. While at the same time in the right trail, the order in which the words are used is governed by the values of π(i) where i is the step within the round and ranges from 0 to 15 (16 steps). Hence, in the right trail and during the first step of round 1, the value of $M[5]$ is used. In step 2, the value of $M[14]$ is used and so on.

The permutation ρ has the effect that two message words which are close in one round are relatively far apart in the next. Similarly, the permutation π has the effect that two message blocks which are contiguous in the left trail will always be at least 7 positions apart in the right trail.

We now summarize the left circular shifts used in each round in table 10 before presenting the details of what happens during a single step operation of the RIPEMD-160 algorithm.

Message blocks	Round				
	1	2	3	4	5
M_0	11	12	13	14	15
M_1	14	13	15	11	12
M_2	15	11	14	12	13
M_3	12	15	11	14	13
M_4	5	6	7	8	9
M_5	8	9	7	6	5
M_6	7	9	6	5	8
M_7	9	7	8	5	6
M_8	11	12	13	14	15
M_9	13	15	14	12	11
M_{10}	14	11	13	15	12
M_{11}	15	13	12	14	11
M_{12}	6	7	5	9	8
M_{13}	7	8	5	9	6
M_{14}	9	7	6	8	5
M_{15}	8	7	9	6	5

Table 10: *Circular left shift of the 16 message words in each round.*

Note that the order in which the shifts are used depends on the order in which the words are used; which in turn depends on the two permutations ρ and π which differ based on the round and on the trail under consideration.
The details of a single step operation of the RIPEMD-160 compression algorithm can be resumed as follows:

$A \leftarrow (A + f(B, C, D) + X_{\rho(j)} + K(j)) <<< s(j) + E$

$C \leftarrow C <<< 10$

where

A, B, C, D, E	=	the five words of the buffer
$f(B, C, D)$	=	primitive logical function used in the current step
$<<< s(j)$	=	circular left shift of the 32-bit argument
$s(j)$	=	the function which determines the amount of rotation for step j
$X_{\rho(j)}$	=	a 32-bit word from the current 512-bit input block
$\rho(j)$	=	permutation function that selects the word to be used in step j
$K(j)$	=	the additive constant used in step j
$+$	=	addition $modulo\, 2^{32}$

After each step and independently of the trail under consideration, the buffer is updated by the following rule:

$A \leftarrow E, E \leftarrow D, D \leftarrow C, C \leftarrow B$, and $B \leftarrow A$

However, Let's underline that the value of the register C is rotated by 10 bits position to the left prior to being assigned to D.

4.4.3 Security of RIPEMD-160 Algorithm

Pre-image resistance

The difficulty of producing any message having a given message digest is of the order of 2160, the same as in SHA-1.

Second pre-image resistance

The difficulty of producing two distinct messages having the same message digest is of the order of 280 operations.

Speed, simplicity and architecture

The addition operation in RIPEMD-160 is perfomed modulo 232, thus it is well suited for a 32-bit architecture. RIPEMD-160 also relies on simple bitwise logical operations, just like MD-5 and SHA-1.

RIPEMD-160 involves 80 steps times two (left and right trails) and must process 160-bit buffer (5 32-bit registers) times two again, it is therefore naturally slower than both SHA-1 and MD5 on the same architecture. But its significant increase in security justifies the reduced performance, and the resulting speed is somewhat

acceptable.

RIPEMD-160's two parallel trails increase considerably the complexity of finding collision between two rounds. This technique usually serves as a basis for finding a collision of the entire compression function.

We also note that RIPEMD-160 resists to known cryptanalysis against both MD5 and SHA-1. This is mainly due to the introduction of the two parallel trails which literally doubles the number of steps performed in the RIPEMD-160 compression algorithm.

RIPEMD-160 is simple to describe and to implement in both software and hardware. RIPEMD-160 uses a little-endian scheme for interpreting a message as a sequence of 32-bit words.

4.4.4 Attacks against RIPEMD-160

As with the SHA-2 family of hash functions developed by NIST, no successful attacks against RIPEMD-160[13] is known. However the designers of RIPEMD-160 envisage that in the next years it will become possible to attack one of the two lines and up to three rounds of the two parallel lines, but that the combination of the two parallel lines will resist today's attacks.

5 Conclusion

We have presented cryptographic hash functions and their main area of application. The focus has been on explaining in great detail what hash functions are, where they can be used and how they are constructed.

As of this writing, a lot is happening in this field. Many recent successful attacks on cryptographic hash functions (including SHA-1[21] and MD5[15]) have urged the U.S. NIST[20] (National Institute of Standards and Technology) to launch a competetion, where public input is sollicited for the development of new hash functions.

Does it mean that these cryptographic hash functions should no longer be used? Maybe not yet, but very soon these cryptographic hash functions will be totally discontinued in favor of stronger ones. In any case, the NIST is already discouraging the use of SHA-1 for certain applications such as digital signature and digital timestamping. The same applies for the usage of MD5 in digital signature scheme.

References

[1] Ralph P. Grimaldi, *Discrete and Combinatorial Mathematics, An applied introduction*, 3rd. Ed, Addison-Wesley Publishing Company, 1994.

[2] William Stallings, *Cryptography And Network Security, Principles and practice*, 2nd. Ed, Prentice Hall 1999.

[3] Alan Bellows, `http://www.damninteresting.com/?p=402`; February 9th, 2006.

[4] Jean-Sebastien Coron, Yevgeniy Dodis, Cecile Malinaud, and Prashant Puniya, "Merkle-Damgård Revisited : how to Construct a Hash Function", `http://www.gemplus.com/smart/rd/publications/pdf/CDMP05.pdf`.

[5] Wikipedia, `http://en.wikipedia.org/wiki/Merkle-Damg%C3%A5rd_hash_function`.

[6] RFC 3174, `http://www.faqs.org/rfcs/rfc3174.html`.

[7] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[8] Bart Van Rompay, *Analysis and Design of Cryptographic Hash Functions, Mac Algorithms and Block Ciphers*; Ph.D. Thesis, june 2004.

[9] Aiden Bruen, David Wehlau, Mario Forcinito, "Hash Functions Based on Sylvester Matrices," Patents Office Kilkenny, September 20th 2001.

[10] Fredrik Skarderud, Ole K. Olsen, Torkjel Søndrol, Ole M. Dahl, *Prosjekt i Kryptologi*, IMT4051 - SHABEIST `http://www.olekasper.no/articles/shabeist.pdf`.

[11] D. R. Stinson, *Some observations on the theory of cryptographic hash functions*, University of Waterloo, Canada, March 2, 2001.

[12] Bart PRENEEL, "Analysis and Design of Cryptographic Hash Functions," Doctoral dissertation, February 2003.

[13] Rüdiger Weis, Stefan Lucks, "Cryptographic Hash Functions, Recent Results on Cryptanalysis and their Implications on System Security," Univeristy of Mannheim.

[14] William Stallings, *Computer Organization and Architecture,* fifth edition, Prenice Hall 1996..

[15] Xiaoyun Wang, Hongbo Yu, "How to Break MD5 and Other Hash Functions," Shandong University, Jinan 250100, China.

[16] Vlastimil Klima, "Finding MD5 Collision - A Toy for a Notebook," Prague, Czech Republic, March 5, 2005.

[17] Jan Anders Solvik, "Hashfunksjoner for bruk i Digitale Signaturer," Hovedfagsoppgave, 20. Oktober 1995, Universitet i Bergen.

[18] Hans Dobbertin, Anton Bosselaers, Bart Preneel, "Strengthened Version of RIPEMD*," 18 april 1996, German Information Security Agency.

[19] Joomla, `http://www.joomla.org/`.

[20] NIST, `http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Jan07.pdf`.

[21] Xiaoyun Wang, Hongbo Yu, Yiqun Lisa Yin "Finding Collisions in the Full SHA-1," `http://people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf`.

[22] Xiaoyun Wang, Hongbo Yu, Yiqun Lisa Yin "Cryptanalysis of the Hash Functions MD4 and RIPEMD," `http://www.cs.ucsd.edu/~bsy/dobbertin.ps`.

[23] NIST, "Secure Hashing," `http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html`.

[24] Hans Dobbertin, "RIPEMD with two-round compress function is not collision-free," Journal of Cryptology 10(1).

[25] Hans Dobbertin, "Cryptanalysis of md5 compress," German Information Security Agency. May 2 1996.

[26] John Talbot, Dominic Welsh "Complexity and Cryptography," An Introduction. Cambridge University Press, 2006.

[27] Robert Churchhouse "Codes and ciphers," Julius Caesar, The Enigma, and the internet. Cambridge University Press, 2004.

[28] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone "Handbook of applied cryptography," CRC Press, 1996.

[29] Bruce Schneier "Applied Cryptography," John Wiley & Sons, 1996.

[30] David Bishop "Introduction to Cryptography with Java Applets," Jones and Bartlett Publishers Canada, 2003.

[31] M. Nandi, and D. R.Stinson "Multicollision Attacks on Generalized Hash Functions," Cryptology ePrint Archive, 2004.

[32] Tom St Denis, "Cryptography for developers,", Syngress Publishing 2007.